STORING AND PREDICTING DYNAMIC ATTRIBUTES IN A WORLD MODEL KNOWLEDGE STORE

By

DANIEL ADAM KENT

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2007

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Storing and Predicting Dynamic Attributes in a World Model Knowledge Store	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Florida,Center for Intelligent Machines and Robotics,Department of Mechanical Engineering,Gainesville,Fl,32611**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The world is an ever-changing, dynamic environment. If robots and other intelligent systems are to find ways to cope with and reason about the world adequately, they must be capable of understanding these dynamic features. This dissertation examines the need for a centralized knowledge store capable of storing information that is both spatial and temporal in nature. The interface of a new and unique architecture to handle the exchange of dynamic information and questions about the future state of that information is presented. A novel algorithm, called the Statistics-Based Nth Order Polynomial Predictor (SNOPP), is also developed which allows state prediction of almost any time-variant data. Each of these contributions is demonstrated through the use of a reference implementation. The author?s reference implementation is done using the Joint Architecture for Unmanned Systems (JAUS), a widely accepted, open robotics architecture developed for use in defense programs. The architecture and predictor are tested using a real-world sensor algorithm deployed on an autonomous vehicle at the University of Florida?s Center for Intelligent Machines and Robotics (CIMAR). Findings and results of a these tests are given which examine the behavior of the architecture and novel prediction algorithm in a variety of scenarios involving different time-variant data types. The Dynamic World Model architecture and the SNOPP algorithm provide significant contributions to the future of robotics. Many robotic problems, including decision making, health monitoring and path planning, stand to benefit from better understanding of the dynamic nature of both the robot and its environment. This dissertation provides a framework in which many of these and other problems may be addressed and summarily solved by future robotic engineers.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**188**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

To my parents, Iris & Bill.
For believing in me when I didn't even believe in myself.

"All our dreams can come true… if we have the courage to pursue them."
Walt Disney

ACKNOWLEDGMENTS

First I would like to thank my family for their unending support of every endeavor I've ever dreamt up. A long time ago I was told to always just do as well as I could; look where it got me today. I would like to extend unending thanks to my Mom, Dad, Brandi, Uncle Steve and especially my hero, Grandpa.

I would also like to sincerely thank my advisor and friend, Dr. Carl Crane, for his incredible support of me and my research. From Gainesville, FL to Barstow, CA, he has always found a way to make my journey more enjoyable. I would also like to thank Dave Armstrong for his endless days of support for me and everything done at CIMAR. Through an Intelligent Ground Vehicle Competition and two DARPA Grand Challenges, I learned that Dave may not always care about aesthetics, but when he does, I'm sure my phone will ring. I would also like to thank the other members of my graduate committee, Dr. Antonio Arroyo, Dr. Warren Dixon, Dr. Eric Schwartz and Dr. Jeff Wit. Their support, critiques and many insights were very helpful to my research.

CIMAR's autonomous ground vehicle work would not be possible without the support of the robotics research group at the Air Force Research Labs at Tyndall Air Force Base, Florida. Special thanks to Walt Waltz, Al Nease and everyone at AFRL for their support of CIMAR.

I would be amiss if I didn't thank the support of the members of the JAUS Working Group, especially Carl Evans and the World Modeling Task Group, for their support of my research. Their many ideas, comments and questions have been invaluable to my research endeavors. I would also like to thank the members of the Experimentation Task Group for the valuable experiences of the many experiments I've been able to participate in over the years.

Lastly, but beyond least, I would like to extend a thank you to the entire CIMAR family, past and present. They have made this journey all the more enjoyable. We may have gotten lost

in the desert along the way, run into a few trees and hit more than our fair share of walls, but we did it together. I would like to especially thank Tom Galluzzo and Bob Touchton for their daily support of my work throughout five long years. I would also like to specifically thank Roberto Montane for everything he's done. Since day one, he's been there for whatever I needed and whenever I needed it. I could not have finished any of this without his help.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

STORING AND PREDICTIONG DYNAMIC ATTRIBUTES IN A WORLD MODEL KNOWLEDGE STORE

By

Daniel Adam Kent

May 2007

Chair: Carl D. Crane, III
Major: Mechanical Engineering

The world is an ever-changing, dynamic environment. If robots and other intelligent systems are to find ways to cope with and reason about the world adequately, they must be capable of understanding these dynamic features. This dissertation examines the need for a centralized knowledge store capable of storing information that is both spatial and temporal in nature. The interface of a new and unique architecture to handle the exchange of dynamic information and questions about the future state of that information is presented. A novel algorithm, called the Statistics-Based N^{th} Order Polynomial Predictor (SNOPP), is also developed which allows state prediction of almost any time-variant data.

Each of these contributions is demonstrated through the use of a reference implementation. The author's reference implementation is done using the Joint Architecture for Unmanned Systems (JAUS), a widely accepted, open robotics architecture developed for use in defense programs.

The architecture and predictor are tested using a real-world sensor algorithm deployed on an autonomous vehicle at the University of Florida's Center for Intelligent Machines and Robotics (CIMAR). Findings and results of a these tests are given which examine the behavior of

the architecture and novel prediction algorithm in a variety of scenarios involving different time-variant data types.

The Dynamic World Model architecture and the SNOPP algorithm provide significant contributions to the future of robotics. Many robotic problems, including decision making, health monitoring and path planning, stand to benefit from better understanding of the dynamic nature of both the robot and its environment. This dissertation provides a framework in which many of these and other problems may be addressed and summarily solved by future robotic engineers.

CHAPTER 1
INTRODUCTION

The world of mobile, intelligent robotics is expanding rapidly. As the shape, size, function and capabilities of these systems change, so does the complexity. One of the more complex functions to be addressed in recent years is that of modeling and understanding more fully the environment in which the robot operates. This dissertation documents the author's work in that area, including background literature material, theoretical details and research results from a reference implementation of the system developed. The focus of the author's work is on modeling dynamic information and predicting the future state of that information.

Two primary contributions are put forth in this document. Firstly, a framework in which a large variety of problems can be addressed and solved is presented, both theoretically and in detail. Second, in the effort to implement and test that framework, a novel algorithm named the Statistics-Based N^{th} Order Polynomial Predictor (SNOPP) is developed with which dynamic attribute prediction is possible; the results of which are presented and discussed.

Motivation and Problem Statement

As robots move out of the lab and into the real world they are forced more and more often to deal with the constraints of that world. One of these constraints which have not historically been dealt with is that the nature of the real world is to exhibit many dynamic characteristics. Historically most mobile robot systems have assumed the world to be relatively static and constant. If a dynamic object was present, it would be treated as a static object with respect to any planning or behavioral efforts.

In more recent work, development has begun of systems which address dynamic elements of the world. Particular effort has been paid to the problem of planning motion within an environment which comprises a combination of static and dynamic elements. Research has also

been conducted in using prediction and estimation techniques in the area of autonomous behavior generation and mission planning.

However, a weakness of each of these approaches is that the data is segregated in such a way that the dynamic information can only be used to accomplish a specific task. The author's work was focused on developing a framework within which any of these and many other tasks can be completed both separately and cooperatively by an autonomous system.

Proposed Solution

To accomplish this task, the author developed a knowledge store capable of storing and querying dynamic information. The knowledge store is responsible for the storage, evaluation and prediction of the future state for any defined dynamic information. This framework supports a variety of possible modeling or prediction algorithms with which a particular item can be modeled. To accomplish this, the sensing or reasoning portion of the robot which seeks to store the information will specify the methodology which the knowledge store should use. The following is an overview of the steps involved in the creation and use of a series of dynamic information. For the sake of this example one should assume the dynamic information is the position of some moving object.

- A sensing element on the robot determines the current position of some moving object. How this is information is determined is not significant to the knowledge store's operation.
- The sensing element creates within the knowledge store the observed object and designates it as "dynamic." It also designates the prediction/estimation algorithm (of the ones possible) which the knowledge store should use in modeling the new object.
- As new position information becomes available, the sensing element updates the knowledge store with the latest information.
- As planning, decision making, or other sensing elements seek to complete their specialized goals, they query the knowledge store for the current or future position of the object. The knowledge store uses the designated estimation algorithm to produce future position estimates.
- The sense and planning elements continue to update and use the information until the object is no longer sensed or of concern to any planning element.

It was beyond the scope of the work presented herein to address evaluation within the knowledge store of the correct selection or appropriateness of the estimation algorithm applied. Instead it is assumed that the sensing element which has identified the information of interest knows the best algorithm which should be applied. For this work, the prediction algorithms available in the knowledge store were defined *a priori* and agreed upon by the sensor and knowledge store developers.

Research Environment

The research documented herein was done at the Center for Intelligent Machines and Robotics (CIMAR) at the University of Florida. CIMAR has been conducting research in autonomous vehicles and robotics for over 15 years and has seen a large amount of success as competitors in the Defense Advanced Research Projects Agency (DARPA) Grand Challenge in both 2004 and 2005 (Figure 1-1 is the NaviGATOR, CIMAR's 2005 entry in the DARPA Grand Challenge). Much of this work is directly related to CIMAR's involvement and support of the robotic activities of the Air Force Research Lab (Tyndall Air Force Base, Panama City, FL). CIMAR has also actively been involved with the Joint Architecture for Unmanned Systems (JAUS) Working Group during the time of author's work. This architecture has provided a solid and reliable foundation on which new ideas and concepts can quickly be implemented and tested in the field.

The NaviGATOR platform is now in use as the primary autonomous test bed at CIMAR. It's a robust and proven platform which includes a variety of sensor systems. It also provides room for ample computing hardware nodes which allow for the rapid development and deployment of new technologies. The system is completely JAUS based and supports a number of essential JAUS components such as the Global Position Sensor and the Velocity State Sensor which are critical to this research (see [1] for more details on these components and JAUS in

general). The NaviGATOR also supports several pre-installed LADAR systems which were used in support of this research. The modularity of the JAUS architecture allows researchers at CIMAR to add and remove non-critical components from the system at will.

CIMAR also has recently developed a vast library of software to support its recent JAUS activities both within the working group's activities and independent research. This software library, written in C, decreases the development time of reliable and efficient JAUS components. The work done in this research made use of these libraries to evaluate a reference implementation. Figure 1-2 shows a dependency diagram of the CIMAR JAUS implementation used in this research.

Figure 1-1 The NaviGATOR, CIMAR's entry in the 2005 DARPA Grand Challenge.

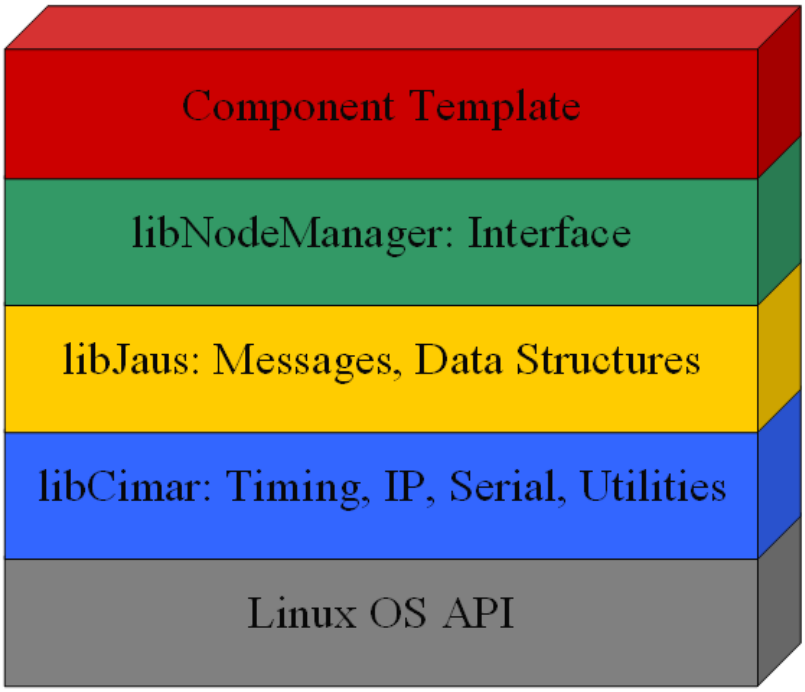

Figure 1-2 CIMAR JAUS Libraries

CHAPTER 2
BACKGROUND AND LITERATURE REVIEW

Work done in this dissertation has laid the foundation for work which can be done in many new and exciting areas within the realm of autonomous and unmanned systems. The Dynamic Environment Concerns section outlines some of the fields in which dynamic objects and attributes are of interest to robotic algorithms. The Motion Prediction and Modeling section provides an overview of some of the many predictive algorithms in use by robotic systems. Lastly, the Database Technologies sections aims to address the methodologies by which this dynamic and spatial information may be stored and made available to a variety of modules simultaneously. A significant contribution of the author's work is that it forms a foundation for future work within the Joint Architecture for Unmanned Systems (JAUS), a standardized robotics architecture. The JAUS and the World Modeling Message Set section examines the structure of the architecture and the current set of world modeling functionality available in JAUS.

Dynamic Environment Concerns

Mobile robots have been slowly moving out of research labs and into the real world since the days of Shakey [2]. In recent years, many of the traditional problems which faced so-called field robotics have been solved with increasing reliability. These include many of the actions along the "sense-plan-act" cycle that most robotic systems demonstrate. However, most of this success has been seen only in the realm of static environments. The real world, and the world researchers ultimately want their robots to operate in, is dynamic. The real world is abundantly full of dynamic parameters and attributes with which robots need to understand and cope. For this reason, in order to sufficiently operate in this real world, the "sense" and "plan" segments of the cycle must be able to reason about and handle these dynamic parameters.

The following are three major areas of interest in which the incorporation of dynamic objects and predictive estimation theories have been applied. In each, the dynamic knowledge has shown an improvement in the system's ability to address its individual concern.

Localization

A primary challenge of any autonomous or semi-autonomous robot is to determine its location within the world. In the realm of outdoor robotics, this is often done through the use of some form of global positioning system (GPS) combined with an inertial navigation system (INS). However, GPS typically does not work for autonomous robots deployed in indoor environments. A popularly accepted approach to the problem of locating a robot within its environment is that of Simultaneous Localization and Mapping (SLAM) which was first outlined in [3]. Since then SLAM has been extensively used in a variety of environments – both 2D and 3D. The primary goal of a SLAM algorithm is to provide an estimate of the robot's position through feature extraction and mapping. Most of this work is based on probabilistic methods which determine a maximum likelihood for the vehicle's position and orientation.

However, much of the research done has operated under the assumption that the robot's environment is static [4]. In [4] is it suggested that if the dynamic objects can be filtered out, the results of the SLAM algorithm will be better. It is also pointed out that while the SLAM algorithms are interested in removing the dynamic obstacles, the detection and tracking of moving objects (DATMO) algorithms are especially interested in exactly the data that is to be filtered. Therefore SLAM with DATMO is presented with favorable results [4].

Other researchers have worked on various other methods to improve SLAM results through careful consideration of dynamic objects. The work has often been applied to problems dealing with the tracking of people [5, 6]. A thorough overview of many of these approaches is presented in [7]. Many of these approaches make use of various probabilistic tracking and

estimation techniques to intelligently estimate which sensor reading should be filtered when considering dynamic objects.

Motion Planning

Planning the future state of an autonomous agent is the essence of the mobile robot problem. While the future state can consist of a number of things, many times the focus on a mobile robot is to plan its behavior, or path, within the environment. In the case of a static or presumed static environment, this problem has to some extent been solved [8]. The solutions to these problems most often align themselves along two fronts, those that find an entire path solution before starting motion, and those that react only to the current world state. So called deliberative and reactive motion planners each have associated strengths and weaknesses (the likes of which are outside the scope of this discussion), however most do not treat dynamic objects in some prescribed fashion. Other work in the area of motion planning has focused on iterative techniques such as D* [9], however these too do not take into account the fundamental nature of dynamic objects. That is predicting and compensating for not only the position or velocity, but the estimated position or velocity of the object at some future point in time.

The most recent work being done in motion planning is beginning to address this three dimensional problem; that is X, Y and time. Initial work in this area concentrated on treating objects as having constant velocity [10]. Later work is beginning to build beyond the constant velocity assumption and addressing so-called Non-Linear Velocity Objects (NLVO) [11]. These approaches and others extend many of the deliberative and iterative planning techniques developed for static problems to allow compensation for moving objects [12-14].

Decision Making and Behavior Control

Motion planning is a significant part of the autonomous mobile robot's job. However, higher level decision making and behavior generation services also face the challenges of a

dynamic and changing environment. Some robotics architectures, notably 4D/RCS [15], dictate the inclusion of prediction and simulation capabilities at each layer of the architecture to facilitate future state estimation. Figure 2-1 illustrates the 4D/RCS architecture and the way prediction and simulation capabilities are layered.

This 4D/RCS architecture has seen use in a number of different autonomous systems. The PRIDE (PRediction in Dynamic Environments) architecture is based on the 4D/RCS scheme and has been developed specifically to address on-road autonomous driving [16]. PRIDE makes use of a multi-resolution, hierarchical architecture to incorporate different prediction methodologies at different levels of the framework. Results from simulation experiments using PRIDE have shown that different predictive routines can be combined to successfully plan at varying time horizons for situation assessment tasks [16, 17].

Similar on-road work has also been done with predictive behavior generation in the field of driver assistance [18]. The work done therein concentrates on an architecture whereby the motivations, goals, plans and probable paths of surrounding vehicles are used to generate predicted situations. Situations are given levels of probability of occurring and all possible situations are fused for decision making. This work also makes use of a database or knowledge store as the fundamental storage and distribution of dynamic data to other modules within the architecture. Figure 2-2 shows the architecture in use by [18] and [19] for intelligent driver assist problems.

While estimations and predictions of future states or events are of significant importance to decisions about the actions of other mobile agents, it is also applicable to the study of the robot itself. MENSA (Mission Effectiveness and Safety Assessment) is an architecture designed to handle primarily health monitoring and contingency resolution [20]. MENSA aids a robot's

ability to reason about itself and arrive at decisions based on its current state, predicted states and mission goals. MENSA is comprised of four modules – sensor monitoring, mission assessment, fault diagnosis, and capability assessment. Figure 2-3 provides a graphical view of the architecture. Central to these modules is a shared knowledge store. In the MENSA architecture, predicted sensor values are performed in the sensor monitoring module.

Motion Prediction and Modeling

There exists a multitude of prediction and modeling algorithms in use for both robotic and non-robotic systems. The following is a survey of various tracking and estimation techniques presented in the literature which have been applied to robots. Many of these approaches rely on some *a priori* knowledge of either the environment or the object being tracked, however some do not.

In [21] a constant velocity model is used to estimate the target's motion in a tracking problem. The constant velocity model uses a short history of the target's motion and extrapolates velocity. To improve the performance of the system, the uncertainty in the direction of the velocity is modeled as a Gaussian distribution. The heading estimate is further improved by incorporating known world information (gaps) into the estimator. The work has shown that the use of velocity estimation compared to similar work without consideration of the dynamics has improved the performance of the tracking algorithm.

In [22] the problem of motion estimation is considered in a fixed environment. The problem is broken into two stages, a learning phase and an estimation phase. In the learning stage, data about the trajectories of real agents moving about the environment is captured. This represents the training data set. The training data is clustered, yielding a series of possible trajectories in the environment. The second phase is the motion estimation phase. In the motion estimation phase, the likelihood that some observed, partial trajectory is part of one of the

trajectories previously learned is evaluated. The estimated motion is then the trained trajectory which has the maximum likelihood. This particular approach depends heavily on the fact that certain paths through a particular environment might be preferred over others. It also embeds an assumption that the environmental constraints which influence these preferred paths tend to remain constant.

In [23] the problem of motion prediction is applied to the RoboCup problem (where a team of robots is challenged with playing soccer against an opposing team of robots). One problem presented in particular is modeling the motion of the soccer ball. In order to predict the behavior of the ball two different models are used. When the ball is rolling freely a neural network is used to predict its motion. When the ball is not rolling freely its motion is estimated through the use of a collision model between it and the robots. The combination of these models was shown to have better results for the prediction of the ball's motion than previous work done using a Kalman filter.

Prediction of the ball for RoboCup has also been studied in [24]. Here a grey prediction algorithm is used. The grey predictor has the advantage of needing to know none of the internal structure or characteristics of the system being observed. Therefore it is able to model and predict the trajectory of the ball sufficiently. This work also implements a switching behavior to the algorithm. Depending on how far the robot is from the ball, the prediction length is modified to reflect the increased time it would take the robot to intercept the ball. Simulated results show an improved performance in the system with the switching grey predictor.

In [25], an algorithm for the prediction of an object's position and orientation is presented which makes use of an autoregressive (AR) model. The complete algorithm for position and orientation estimation is presented in separate parts, first dealing with the translational motion

then the rotational. The results presented show good performance of the system in estimating motion both with varying and uniform acceleration.

The prediction work done in [26] focuses primarily on estimating the motion of other intelligent agents. Here an "Intelligent Prediction" algorithm is developed. This approach takes into account the dynamic and kinematic characteristics observed along with some a priori knowledge of the agent's goals and behaviors.

In [23] Kalman filtering techniques are presented to solve the problem of estimating the position and orientation of moving obstacles. An advantage of the Kalman filtering approach is that the estimator begins with the first step, whereas most other approaches require some minimum data history to yield favorable estimates. Two Kalman filters are developed, one for translational estimation and the other for rotational.

Work done in [27] focuses mainly on the use of predictive motion in building a motion planner. However, it does outline a method for estimated motion based on a Polynomial Neural Network (PNN). The PNN is seeded with the moving object's position at the current and previous time steps and the output is the estimated position of the object at the next time step. As in most neural networks, the performance of the system is very dependent on a large set of training data.

Database Technologies

An important contribution of this work is to take the storage and prediction elements of spatial information and co-locate them into one service on the mobile platform. Current database systems, both object-relational and relational, can be extended to support spatial and temporal capabilities (see Table 2-1 for more details). Recent advances in spatiotemporal and moving objects databases provide a framework for the query and storage of objects that move spatially within the environment.

Spatial Database Technologies

A spatial database is a relational or object-oriented database which has been enhanced to support spatial data and perform spatial operators on that data (for a through technical discussion of spatial databases and their operators see [28]). Spatial data is usually divided into two primary categories, raster and vector. The focus of the work in this research is in the storage and interpretation of vector representations. Vector data is typically divided into one of three types: points, lines and polygons. In general, these objects are not restricted to 2D representations and can include a mixture of planar and non-planar data. An important aspect of spatial database is not only the storage of spatial data types, but the ability to perform logical operations on that data. Typically these operations include at minimum relational operations between entities such as touches (any point of both entities are equal), intersect (entities intersect such that there is a set of collective points shared), or within (one entity completely encapsulated within the interior of another). Figure 2-4 describes these operators.

An important feature of most commercial spatial databases is that of spatial indexes. A spatial index makes use of the spatial relationships among data members to improve the performance of queries and spatial operators [28]. Table 2-1 lists several commonly available relational and object-relational database systems and the various spatial options available for each. Several other factors are presented in Table 2-1 including 3D data support, raster data support, compliance with the standard outlined by the OpenGIS Consortium (OGC) in [29] and relative system costs.

Moving Objects Databases

Recent work in DBMSs has concentrated more on the moving object problem. In a Moving Object Database (MOD), information about geometries that change over time are stored. The

MOD is an extension to the spatial database and the temporal database [30]. MODs are comprised primarily of two distinct parts, the query language and the spatio-temporal data types.

Early work on MODs could not make use of DBMS systems due to limitations on indexing and data types. Two significant efforts in the early development of MODs sought to overcome these limitations, namely the Databases fOr MovINg Objects (DOMINO) project [31-33] which implements the Moving Objects Spatio-Temporal (MOST) data model [34, 35]. The MOST data model introduced the concept of a *dynamic attribute*. This is an attribute whose value changes with respect to time without being explicitly updated [34]. These efforts sought to solve many of the fundamental DBMS questions in dealing with moving objects. To simplify the problems, these efforts made use of a number of assumptions [30].

Several of the assumptions hinder the technology's use in robotics. Paramount to these is an assumption that the moving object will update the database with new information whenever the object's position deviates by some threshold. Based on this assumption, uncertainty in the object's position is based on knowing that it is at most "threshold" distance away from its last known position. The MOST model also supports two methods to model the object's motion, first through the use of a motion vector. This vector can be any function with respect to time that describes the future motion of the attribute it describes. Higher-level information can also be used. In the case of vehicles, it is more realistic to assume that the vehicle travels along some road network. If the possible road network is known, then the motion of the vehicle can be constrained to that network [30].

The significant work done to date in MODs concentrates on the storage, modeling and querying of moving objects, including work on uncertainty analysis [36]. However, little to no work has been done to include within the MOD the ability for the database to develop it own

model of the attribute and reason about its future. Instead a MOD relies on a close relationship with the moving object itself to provide that information.

JAUS and the World Model Message Set

The Joint Architecture for Unmanned Systems (JAUS) is an effort to develop an upper level design for the various interfaces in the realm of unmanned systems. The initiative started in 1998 when the Office of the Secretary of Defense charted what was then the Joint Architecture for Unmanned Ground Vehicles Working Group (JAUGS WG). JAUGS has since transitioned to encompass the realm of all unmanned systems. It is the goal of the working group to develop a framework which

- aids in procurement of robotics systems by ensuring mutual compatibility,
- encourages industry competition without fear of being locked in to proprietary solutions,
- enables developers to focus on application needs rather than basic infrastructure, and
- reduces the burden of technology transfer between programs.

JAUS defines a component-based messaging architecture which defines the data format and message specifications for communicating between different computing nodes and processes. It defines an architectural layout of System, Subsystem, Node and Component which correspond, respectively, to a combination of robotic resources, a particular robot or control unit, a computing resource and a software process. Figure 2-5 shows the JAUS topology. Various components are defined in the JAUS Reference Architecture (RA) for well-defined robotic software roles [1]. These include global position, velocity feedback, actuator control and configuration management. The RA includes a definition of various data types used to communicate between components. It also includes a set of messages which define the way in which information is passed between components and the nature in which this communication takes place.

Much work has been done in recent years to progress JAUS into emerging areas of robotic research. One key area of interest to the JAUS Working Group (WG) has been in the area of World Modeling. What follows is the definition of a World Model as adopted by the JAUS standard.

> The world model is the intelligent system's best estimate of the state of the world. The world model includes a database of knowledge about the world, plus a database management system that stores and retrieves information. The world model also contains a simulation capability that generates expectations and predictions. The world model provides answers to requests about the present, past, and probable future states of the world. The world model provides this information service to the behavior generation system element in order to make intelligent plans and behavioral choices. It provides information to the sensory processing system element to perform correlation, model matching, and model-based recognition of states, objects, and events. It provides information to the value judgment system element to compute values such as cost, benefit, risk, uncertainty, importance, and attractiveness. The world model is kept up to date by the sensory processing system element. [37]

As can be seen, one of the primary goals of the World Model, as defined above, is to provide a "simulation capability." However, work done to date within JAUS deals solely with the query and storage of static geospatial data [38]. The work done in this dissertation focuses on vector data representation, therefore the area of interest in the JAUS documentation is that which deals with the World Model Vector Knowledge Store. The messages and capabilities therein support the following capabilities:

- Create point, line or polygonal object(s). These objects can contain a number of feature classes [38] and each feature class may contain a single feature class attribute.
- Delete an object or a group of objects in an area.
- Query for objects by region, id, feature class, attribute or any combination thereof.

This core set of query and storage functionality, along with the developed spatial data types provided the foundation for the implementation detailed in Chapter 4. The JAUS World Model message set also establishes a set of terminology including feature class and feature class

attribute which are used extensively in the author's implementation. More details about these can be found in [38].

While much of the basic functionality is retained, many of the existing JAUS messages needed to be changed or extended to support the architectural requirements. The modified JAUS message set, as used by the author, is included as Appendix A.

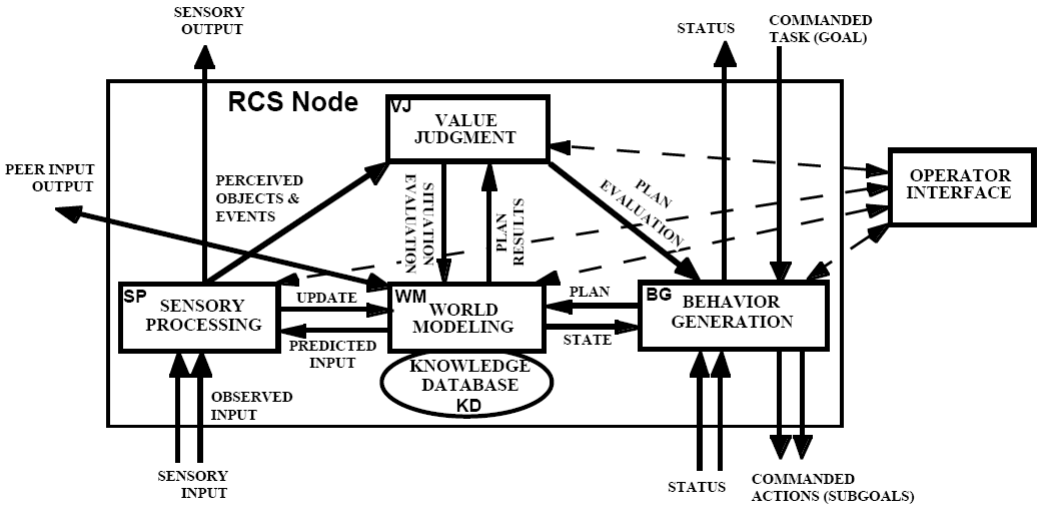

Figure 2-1 Illustration of the 4D/RCS architecture to include World Modeling in direct support of sensing, behavior generation and value judgment activities. [15]

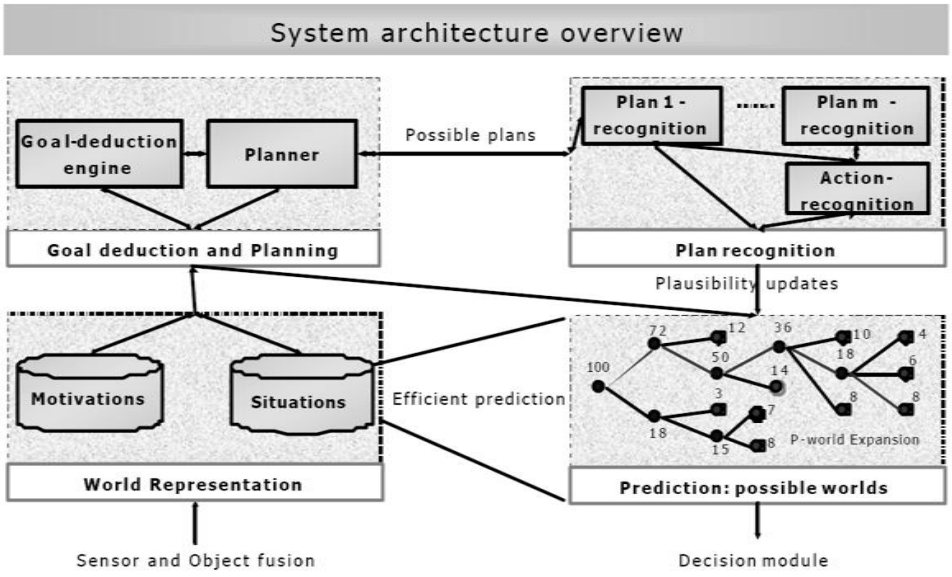

Figure 2-2 System architecture for decision making in driver assistance system [19]

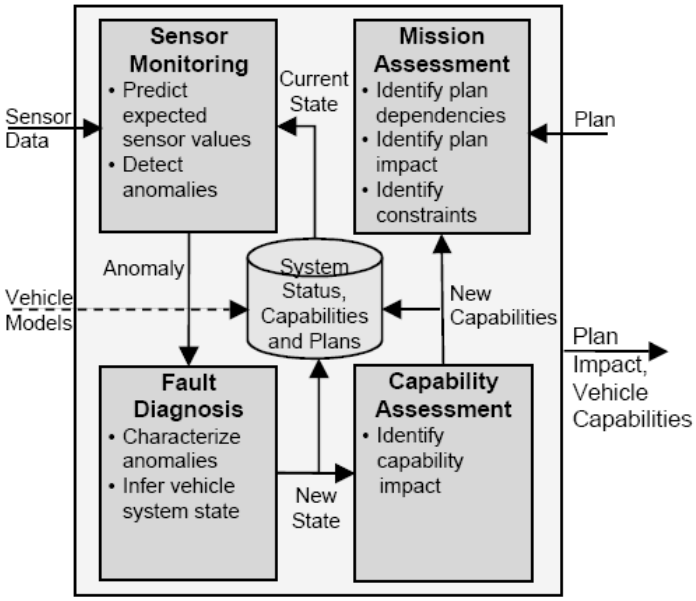

Figure 2-3 MENSA architecture for robot self awareness [20]

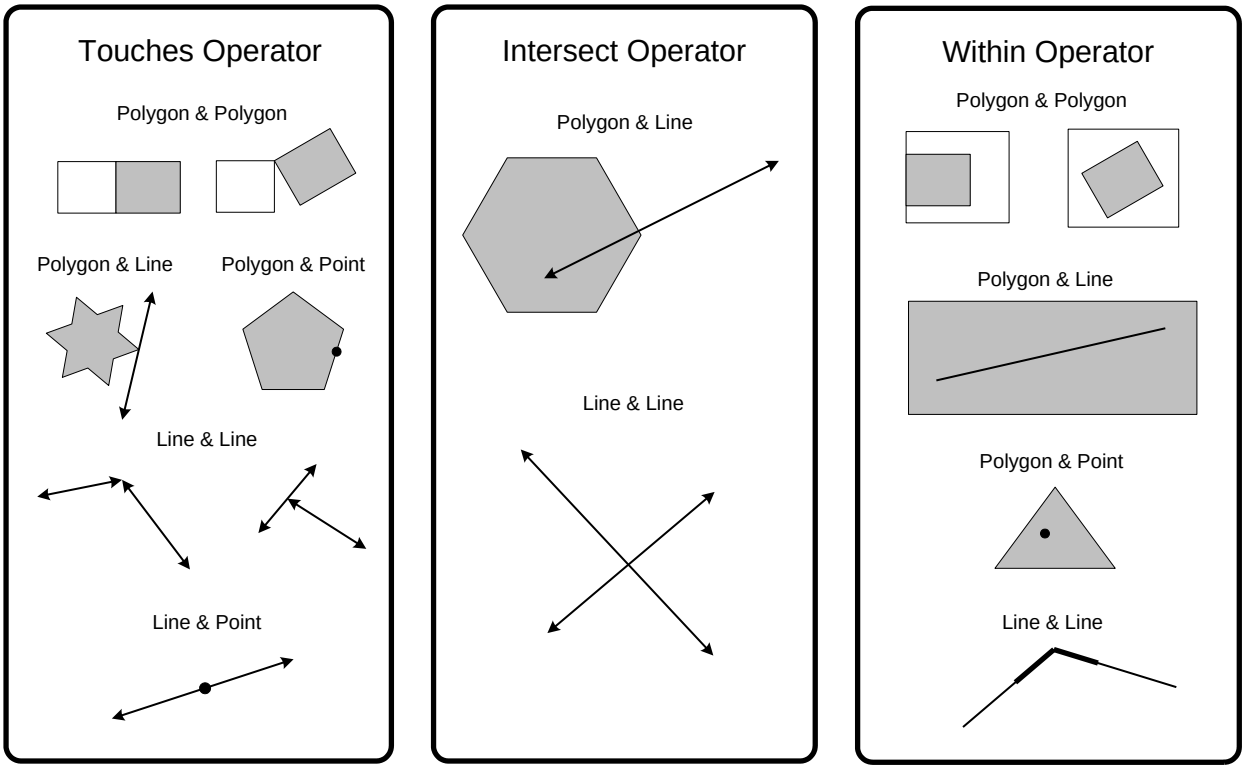

Figure 2-4 Spatial operators Touches, Intersect and Within.

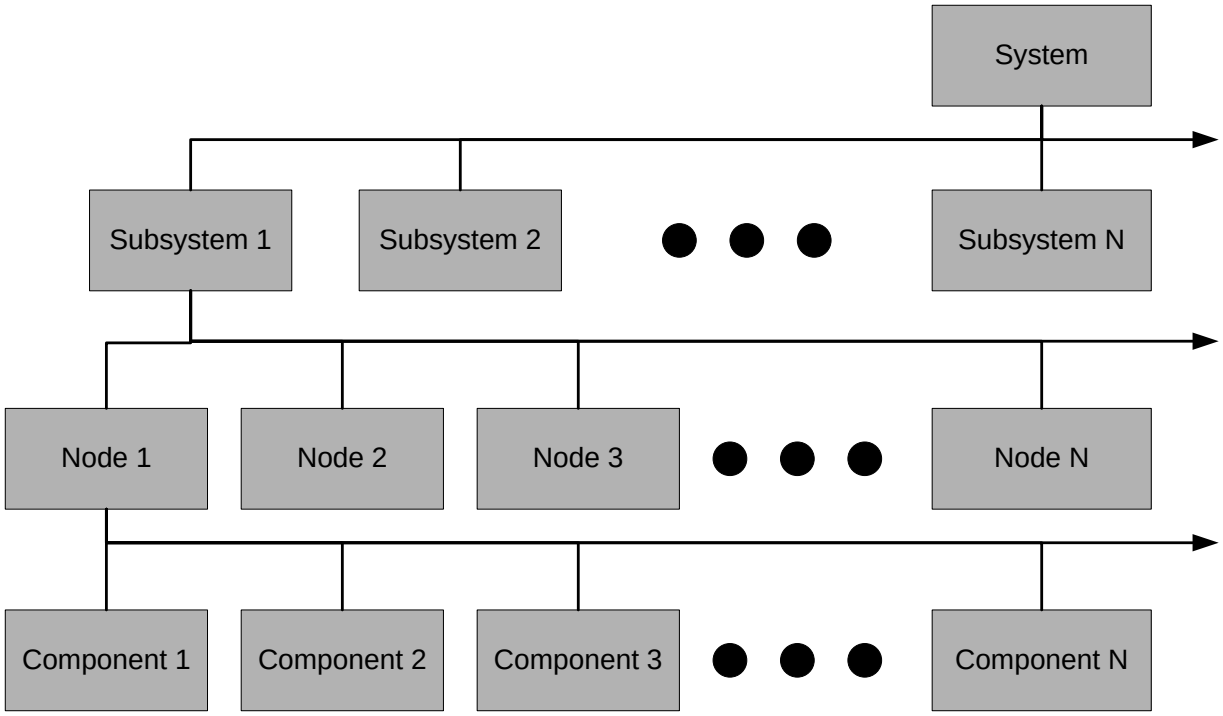

Figure 2-5 JAUS System Topology

Table 2-1 Common Spatial Database Management Systems (DBMS) [39-47]

DBMS	OS	Spatial Extension	OGC Compliance	Spatial Index	Coordinate System	3D	Raster	Cost
Oracle	Linux, Windows	Oracle Spatial	Compliant	●	●	●	●	$$$$$
		Oracle Locator	Compliant	●	●	●	-	$$
		ArcSDE	Compliant	●	●	●	●	$$$
DB2	Linux, Windows	DB2 Spatial Extender	Compliant	●	●	-	-	$$
		ArcSDE	Compliant	●	●	●	●	$$
MS Access	Windows	GeoMedia	-	●	●	●	-	$
MS SQL Server	Windows	SpatialWare	Implementing	●	●	●	-	$
		GeoMedia	-	●	●	●	-	$
		ArcSDE	Compliant	●	●	●	●	$$
Informix	Linux, Windows	Spatial Datablade	Compliant	●	●	●	-	$$
		ArcSDE	Compliant	●	●	●	●	$$$
PostgreSQL	Linux, Windows	PostGIS	Compliant	●	●	●	-	Open-Source
MySQL	Linux, Windows	Native	-	●	-	-	-	Open-Source

CHAPTER 3
THEORETICAL APPROACH

The storage and querying of dynamic information is a challenging problem. In this chapter the author describes two parts of the unique problem. Firstly, the architectural requirements of a centralized knowledge store are discussed and a solution proposed. Following that, a unique prediction algorithm for estimating the future state of information without a priori knowledge of the behavior of the system is described. Through the combination of the architecture described and use of various prediction methods like the one described herein, a vast array of dynamic and temporal knowledge store problems may be solved.

Spatiotemporal World Model Architecture

The use of knowledge stores as centralized repositories for information is not a new concept in either computer science or robotics. There is a long tradition of knowledge stores such as databases being used in applications ranging from financial to stock control. Also, in robotics, common, centralized knowledge representations have been used successfully in a number of robotic systems. However, few of these systems have attempted to cope with the various demands of both spatial and temporal knowledge. Further, providing predictive behavior about the future state of information is something only recently explored by the moving objects database (MOD) researchers.

In developing an architecture for use on a robotic system, several constraints and concerns are of importance. Firstly is the flexibility of the interface. The kind of spatiotemporal data most often encountered on an intelligent robotic platform is from some form of sensing capability. It is then appropriate to design the interface to the knowledge store in such a way as to cater to the capabilities of most sensing systems. Secondly, the architecture should scale with the scope of the robot and its mission. From simple vacuuming robotic systems for home use to a fleet of

robotic platforms deployed together, the spatiotemporal knowledge store should provide a solution that suits the needs of both. To those ends, the architecture presented here provides a flexible yet robust interface which allows the robotic system designer to make best use of the information available on the system. It provides capabilities to create objects, modify them, delete them and ask questions about their past, present and future state. Each object is assigned a unique object id for identification purposes either by the knowledge store or the client. Objects may have associated with them both geospatial information and attribute information. The geospatial information describes the geometry, position and orientation of an object in the context of the world. The attribute information can be used to hold any other significant information about the object which may include but is not limited to color, velocity, height, classification and/or name.

A novel feature of this architecture is the capacity for multiple estimation techniques for object and attributes prediction. Rather than attempt to apply a single prediction technique to all objects, the knowledge store is flexible in allowing the client to specify which technique to use. Current work has focused on having a collection of prediction techniques defined *a priori* which a knowledge store client can select from in creating or modifying objects and their attributes. This capability extends the flexibility and scalability of the architecture by allowing particular prediction techniques to be created and scaled to the particular application of interest; without the need to modify or change the overall architecture and message set.

Figure 3-1 shows the various input and output messages for the architecture. Each message and its various fields are described below in detail.

Request Object Id Block Message

- REQUEST ID
- BLOCK SIZE

The **Request Object Id Block** message is used by the knowledge store client to request a range of object ids which it may use. The REQUEST ID is used by the knowledge store to track any response to the client. The BLOCK SIZE field indicated the number of object ids the client is requesting.

Assign Object Id Block Message

- REQUEST ID
- BLOCK SIZE
- OBJECT IDS

The **Assign Object Id Block** message is sent in response to a **Request Object Id Block** message. This message includes the REQUEST ID of the original request. The BLOCK SIZE field is the size of object ids allocated by the knowledge store for use by the requesting client. The OBJECT IDS field is a list of the object ids allocated.

Create Object(s) Message

- REQUEST ID
- OBJECT COUNT
- OBJECT ID *(OPTIONAL)*
- OBJECT ESTIMATOR TYPE *(OPTIONAL)*
- OBJECT TIME STAMP *(OPTIONAL)*
- OBJECT GEOSPATIAL INFORMATION
- OBJECT ATTRIBUTES INFORMATION

The **Create Object(s)** message includes allows the client to create one or more unique objects within the knowledge store. The REQUEST ID field is used to track the creation request and is used in any response from the knowledge store. The OBJECT COUNT field is provided to indicate how many objects are included in the message. Each object is constructed from set of fields including OBJECT ID, OBJECT ESTIMATOR TYPE, OBJECT TIME STAMP, OBJECT

GEOSPATIAL INFORMATION and OBJECT ATTRIBUTES. The OBJECT ID field is optional. It is provided when the client would like to create an object with a particular unique object id. The OBJECT ESTIMATOR TYPE field is also optional. It indicates which of the available estimation algorithms available should be used to estimate the geospatial position of the object for queries about its future state. If this field is not present, it is assumed that the object is to be considered static by the knowledge store. The OBJECT TIME STAMP field is optional. This field is used to indicate the time at which the geospatial information about the object was measured or otherwise known to be valid. The OBJECT GEOSPATIAL INFORMATION is a collection of all the information needed to completely define the geospatial configuration of the object. Lastly, the OBJECT ATTRIBUTES INFORMATION is a list of attributes which are attached to the object. It is assumed that each attribute can also be declared as either static or dynamic with an appropriate estimator and timestamp.

Confirm Creation Message

- REQUEST ID
- CREATION RESULT
- OBJECT IDS

The **Confirm Creation** message is sent by the knowledge store to the client following the receipt of a **Create Object(s)** message. This message includes the REQUEST ID of the original creation. The CREATION RESULT field is used to indicate the success or failure of the creation request. For each object in the original request, the appropriate OBJECT ID (either assigned by the knowledge store or as requested) is included.

Modify Object(s) Message

- REQUEST ID
- OBJECT COUNT

- Object Id
- Object Estimator Type *(optional)*
- Object Time Stamp *(optional)*
- Object Geospatial Information
- Object Attributes Information (*optional*)

The **Modify Object** message is used by the knowledge store client to modify or add information about an object in the knowledge store. The Request Id is used by the knowledge store in its response to track the original request. The Object Count field is used to indicate the number of objects included to be modified. The Object Id field indicates which object in the knowledge store that the client would like changed or updated. Like the **Create Object(s)** message, this message includes information about the object which should be changed or updated. The client is given the opportunity to change the Object Estimator Type, update the Object Time Stamp and update or replace the Object Geospatial Information and Object Attributes Information.

Confirm Modification Message

- Request Id
- Modification Result

The **Confirm Modification** message is sent by the knowledge store in response to a **Modify Object** message. This message includes two fields, the Request Id of the original modification message and a result. The Modification Result is used to indicate success or failure of the request.

Delete Object(s) Message

- Request Id
- Object Ids *(optional)*
- Deletion Region *(optional)*
- Deletion Attributes *(optional)*

The **Delete Object(s)** message is sent to the knowledge store from a client who wishes to delete one or more objects from the knowledge store. The REQUEST ID is used to track the input for any response. There are 3 ways objects can be identified for deletion. The first is to identify the unique OBJECT ID of the object(s) to be deleted. The second method is to provide a DELETION REGION. Here the client provides a geometric region to the knowledge store; all objects within that region (either wholly or partly) will be deleted. The last method is to provide a list of DELETION ATTRIBUTES. This will cause the knowledge store to remove any object which matches all the attributes provided. These various methods can be combined to form more complex deletion requests. Such combinations result in logical conjunctions and could allow deletion of a particular object only if it is inside the deletion region or removal of all objects in a region that match a particular attribute value. Through combinations of the three methods, a large number of flexible deletion commands can be constructed.

Confirm Deletion Message

- REQUEST ID
- DELETION RESULT
- OBJECT IDS

The **Confirm Deletion** message is sent by the knowledge store in response to a **Delete Object(s)** message. This includes the REQUEST ID field of the original deletion request for tracking purposes. The DELETION RESULT indicates success or failure of the request. The OBJECT IDS field is a list of the objects successfully removed from the knowledge store.

Query Object(s) Message

- REQUEST ID
- HISTORY SIZE *(OPTIONAL)*

- Object Ids *(optional)*
- Query Region *(optional)*
- Query Attributes *(optional)*

The **Query Object(s)** message is used by a client to find information about the current or past state of one or more objects in the knowledge store. The Request Id is used to track the query and identify the response from the knowledge store. The History Size field is optional. This field indicates the time period of object history to include in the response. There are three ways objects can be identified for inclusion in the response. The first method is to explicitly identify the object(s) through the use of their Object Ids. The second method consists of identifying a Query Region. This defines a geometric region in which all objects within (either wholly or partly) will be included in the result. Lastly, specific Query Attributes can be included. Any object which matches all the included attributes will be included in the result. Like the **Delete Object(s)** message, these fields can be combined to provide more flexible queries. Combinations are considered to be logical conjunctions. This allows queries such as a specific object if it is inside some region or all objects with a particular attribute value.

Report Object(s) Message

- Request Id
- Query Result
- History Size *(optional)*
- Object Count
- Object Id
- Object Estimator Type *(optional)*
- Object Time Stamp *(optional)*
- Object Geospatial Information
- Object Attributes Information

The **Report Object(s)** message is sent by the knowledge store in response to a **Query Object(s)** message. This message contains the results, if any, of the presented query. The

REQUEST ID is used to identify to the client which query this is the response to. The QUERY RESULT field is used to indicate the success or failure of the original query. The HISTORY SIZE field is optional and is included if the original query included it. The OBJECT COUNT field indicates how many unique objects are included in the report. For each object included, the OBJECT ID, OBJECT ESTIMATOR TYPE, OBJECT TIME STAMP, OBJECT GEOSPATIAL INFORMATION and OBJECT ATTRIBUTES INFORMATION are included as appropriate. If a history of data is included, each of the OBJECT TIME STAMP, OBJECT GEOSPATIAL INFORMATION and OBJECT ATTRIBUTE INFORMATION will include a collection of values which constitute the history of object.

Query Object(s) Future State Message

- REQUEST ID
- QUERY TIME
- OBJECT COUNT
- OBJECT ID
- OBJECT ATTRIBUTES *(OPTIONAL)*

The **Query Object(s) Future State** message is sent to the knowledge store by a client which is interested in the future state of some object(s). The REQUEST ID is used by the knowledge store to track the response. The QUERY TIME field is used to indicate the point in time at which the future state of the object is of interest. The OBJECT COUNT field indicates how many objects are included in the query. Each object queried includes the unique OBJECT ID and which OBJECT ATTRIBUTES (if any) are to be included in the response.

Report Object(s) Future State Message

- REQUEST ID
- QUERY RESULT
- OBJECT COUNT
- OBJECT ID

- Object Time Stamp
- Object Geospatial Information Estimate
- Object Attributes Information Estimate *(optional)*

The **Report Object(s) Future State** message is sent by the knowledge store in response to a **Query Object(s) Future State** message. The Request Id field is used to indicate which originating query this response is for. The Query Result field indicates the success or failure of the query. Object Count indicates how many objects are included in the report. Each object includes its unique Object Id, the Object Time Stamp for the future state, the Object Geospatial Information Estimate and if requested, the Object Attributes Information Estimate.

Query Geospatial Bounds Message

- Request Id
- Object Attributes Information *(optional)*

The **Query Geospatial Bounds** message is used by the knowledge store client to ask for the geometric bounds of the knowledge store. The Request Id is used to identify the query and its response. The Object Attributes Information field is optional. This field can be used to specify one or more attributes to filter the results. This causes the knowledge store to return the bounding geometry of only the objects that match the given attributes.

Report Geospatial Bounds Message

- Request Id
- Query Result
- Boundary Geospatial Information
- Object Attributes Information *(optional)*

The **Report Geospatial Bounds** message is sent in response to a **Query Geospatial Bounds** message. This message includes four fields. The REQUEST ID field is used to identify which query this result is in response to. The QUERY RESULT indicates the success of failure of the input query. The BOUNDARY GEOSPATIAL INFORMATION defines the geometry of the knowledge store's boundary based on the query. If the result has been filtered by some attributes, the OBJECT ATTRIBUTES INFORMATION is included in the response.

Query Temporal Bounds Message

- REQUEST ID
- OBJECT ATTRIBUTES INFORMATION *(OPTIONAL)*

The **Query Temporal Bounds** message is used by the knowledge store client to ask for the temporal bounds of the knowledge store. The REQUEST ID is used to identify the query and its response. The OBJECT ATTRIBUTES INFORMATION field is optional. This field can be used to specify one or more attributes to filter the results. This causes the knowledge store to return the upper and lower timestamp of the objects that only match the given attributes.

Report Temporal Bounds Message

- REQUEST ID
- QUERY RESULT
- TEMPORAL INFORMATION
- OBJECT ATTRIBUTES INFORMATION *(OPTIONAL)*

The **Report Temporal Bounds** message is sent in response to a **Query Temporal Bounds** message. This message includes four fields. The REQUEST ID field is used to identify which query this result is in response to. The QUERY RESULT indicates the success of failure of the input query. The TEMPORAL INFORMATION defines the upper and lower time stamp of the knowledge

store's boundary based on the query. If the result has been filtered by some attributes, the OBJECT ATTRIBUTES INFORMATION is included in the response.

The Statistics-Based Nth Order Polynomial Predictor

The methods by which the future state of a value can be estimated, or predicted, are multitude. Many of the more popular methods are covered within the literature review provided in Chapter Two. As discussed there, many of those methods, including Kalman Filters and most Neural Networks, incorporate a model of the object to aid in the prediction of the algorithm. For the author's work, it was desired to find an algorithm that was generic enough to be applied to a very large variety of data without *a priori* knowledge of the system model.

To that end, the author has devised a unique algorithm called the Statistics-Based Nth Order Polynomial Predictor (SNOPP). SNOPP is a novel combination of real-time data analysis and statistics to generate a polynomial which can be used to estimate the future state of any continuous time-variant data. That is, for a small change in time, the change in the output is small. Systems exhibiting discontinuous behavior can be modeled, however large errors may be observed around any discontinuous point.

SNOPP works to find a polynomial which best fits the trend of a given data set. Stated formally, the data set is the matrix **D** which has size $n \times 2$:

$$\boldsymbol{D} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}. \tag{3.1}$$

The algorithm works by analyzing a given data set repetitively, attempting to fit a polynomial to the data at a series of different orders (1st, 2nd … nth). It also varies the length, or history, of information. By varying the history size, SNOPP can achieve stable results for long data trends, yet quickly react to changes in the data set. Three parameters govern the solution

space of the algorithm, a maximum order value, the window size and window count. The maximum order value is an upper limit on the order to which the algorithm will attempt to fit a polynomial. Window size is the number of data points which will be analyzed per solution iteration. Window count is the number of data windows to solve, which is the number of solutions to iterate through. Figure 3-2 shows the relationship between the window size and window count parameters for some hypothetical data set. The window size parameter should always be set equal to or greater than the maximum order value in ensure convergence of the solution of the polynomial. Once the three parameters are set, the algorithm continues as follows.

SNOPP independently analyzes subsets of the original data set which are defined by the window size and window count. For each subset, it seeks to find the best fit polynomial of up to the maximum order. First, a polynomial of the highest possible order (let *k* = maximum order) is fit to the data using the least-squares fit method. This polynomial is defined as,

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_n x^k. \tag{3.2}$$

The least-squares method is based upon minimizing the sum of the residuals at each data point. The residual equation is given as,

$$\mathcal{R}^2 = \sum_{i=1}^{n} [y_i - (\beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_k x^k)]^2. \tag{3.3}$$

The solution then, is to minimize $\mathcal{R}^2$ with respect to each β,

$$\frac{\partial(\mathcal{R}^2)}{\partial \beta_i} = 0, for\ i = 1, \ldots k. \tag{3.4}$$

The polynomial can also be described in matrix form as,

$$\hat{\boldsymbol{y}} = \mathcal{X}\hat{\boldsymbol{\beta}}. \tag{3.5}$$

where: $\hat{\boldsymbol{y}} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ $\hat{\boldsymbol{\beta}} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{bmatrix}$ and $\mathcal{X} = \begin{bmatrix} 1 & x_1 & \cdots & x_1^k \\ 1 & x_2 & \cdots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k \end{bmatrix}$

In this form the solution for $\hat{\boldsymbol{\beta}}$ can be found by premultiplying by $\mathcal{X}^t$ as,

$$\mathcal{X}^T\hat{\boldsymbol{y}} = \mathcal{X}^T\mathcal{X}\hat{\boldsymbol{\beta}}, \tag{3.6}$$

which can be solved numerically or inverted directly if $\mathcal{X}$ is well-formed, resulting in:

$$\hat{\boldsymbol{\beta}} = (\mathcal{X}^T\mathcal{X})^{-1}\mathcal{X}^T\hat{\boldsymbol{y}}. \tag{3.7}$$

This yields a *k*-order polynomial that is the best fit to the given data sub-set. However, it is plausible that one or more of the order terms are not statistically significant. That is, while the algorithm may have attempted to fit a 5th-order polynomial and found said solution, the best fit to the data may actually be a 2nd-order polynomial. SNOPP uses a Type I Sum of Squares statistical test to determine exactly what order polynomial best approximates the given data. The Type I Sum of Squares evaluates the difference between a polynomial and its lower-order equivalent for each polynomial up to the maximum order. For each test, the null hypothesis is that $H_0: \beta_i = 0$ while the alternative hypothesis is $H_a: \beta_i \neq 0$. This test is done using the following as the F Value:

$$F_i = \frac{(SS_{Model})_i - (SS_{Model})_{(i-1)}}{MSE}. \tag{3.8}$$

Here *MSE* is the Mean Squared Error of the original *k*-order polynomial and is found by

$$MSE = \frac{\sum_{i=1}^{k}\sum_{j=1}^{n}\left(y_{ij} - \bar{y}\right)^2}{n - (k+1)} \tag{3.9}$$

where $\bar{y}$ is the mean of the dataset.

$(SS_{Model})_i$ is defined as the Sum of Squares of the model or residuals and is calculated as

$$(SS_{Model})_i = \sum_{j=0}^{n} \left(\widehat{\boldsymbol{\beta}}_{\boldsymbol{i}} \widehat{\boldsymbol{x}}_j - \bar{y}\right)^2. \quad (3.10)$$

where $\widehat{\boldsymbol{x}}_j = \begin{bmatrix} 1 & x_j & \cdots & x_j^i \end{bmatrix}$

It is important to note here that $\widehat{\boldsymbol{\beta}}_{\boldsymbol{i}}$ is the value obtained from a least squares fit of a polynomial of *i*-order. That is to say a polynomial must be fit to the original dataset for every order 1 to *k*.

Using equation (3.8) a value for F_i can be found. This value is evaluated against the standard F-Distribution, which is a one-tailed continuous distribution. The F-Distribution is evaluated with two degrees of freedom. The first degree of freedom is equal to 1 because this test is always evaluating an equation with one more term of variance than the former, $df_1 = 1$. The second degree of freedom is equal to the degree of freedom of the *MSE* value, or $df_2 = n - (k + 1)$. The alternative hypothesis $(H_a: \beta_i \neq 0)$ is rejected if the probability of it being untrue is above some critical value. Typically a very large probability is applied to this test such as 20%. If the test reveals that a particular β_i has a high probability of being equal to zero, then that term and all higher-level terms are considered zero.

This yields a value for exactly what order polynomial best fits the given data set (let *p* equal this order). This yields a polynomial of order *p* for the given window size. SNOPP then repeats this polynomial fit and analysis a total of window count times. Each time, the data set grows by window size data points. This provides a number possible polynomials, each statistically determined to be the best fit for their given data set. The algorithm then selects the polynomial which has the lowest order solution. In case of a tie, the solution with the largest data set is used.

The lowest order polynomial is chosen because higher-order polynomials tend to fit data well within the dataset, but diverge wildly when used for extrapolation, which is the goal of the

future state estimator. The data set with the largest history is used because with larger data sets provide more statistical significance and a more stable trend for data extrapolation. By combining small window size with a large number of window counts, long, stable trends in the data can be accommodated, but changes in that trend can be quickly detected and adjusted to by the smaller datasets which represent the most recent history of the data.

A numeric example of the SNOPP algorithm is included in Appendix C of this dissertation for clarity.

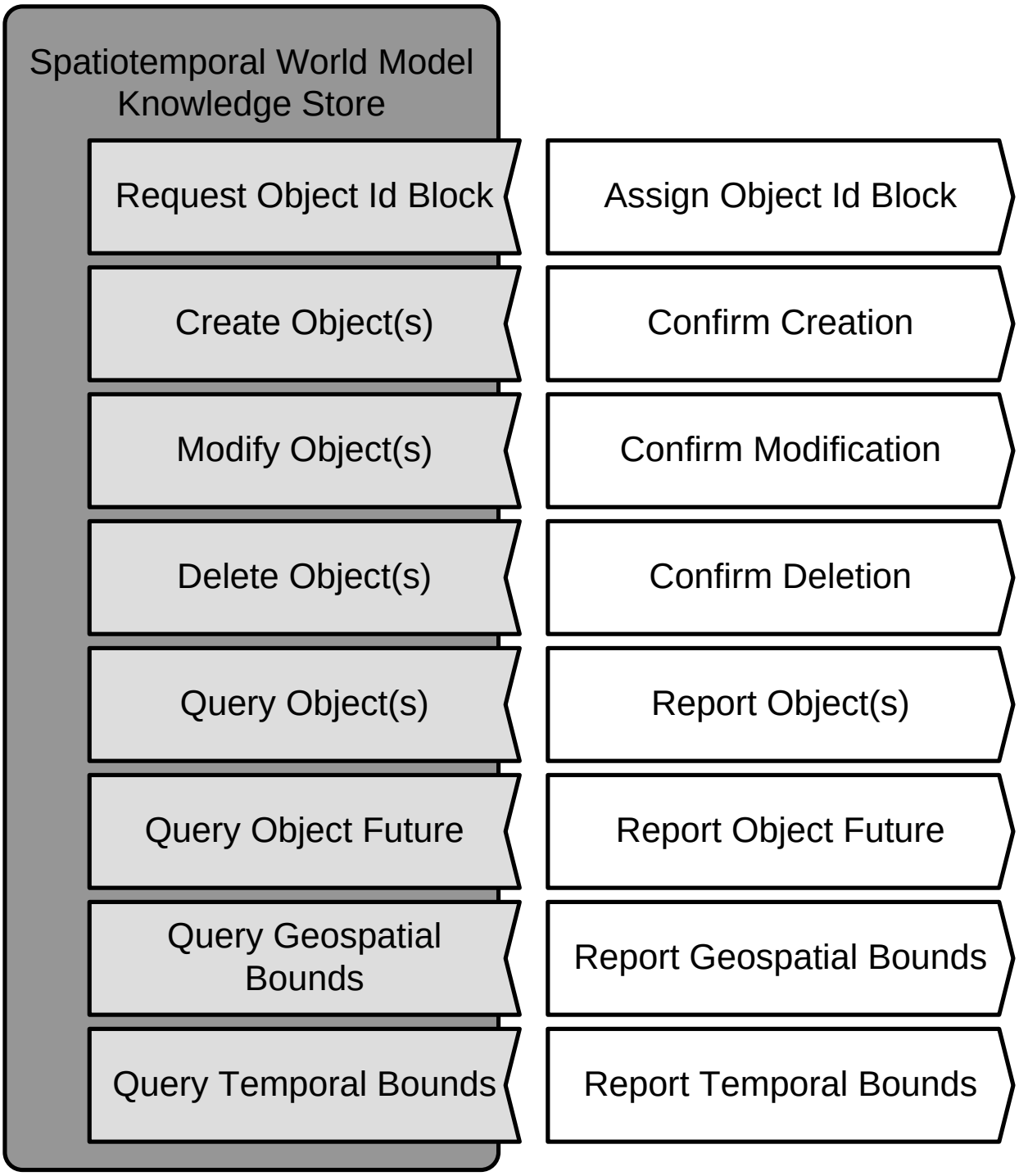

Figure 3-1 Overview of the input and output messages associated with the Spatiotemporal World Model Architecture.

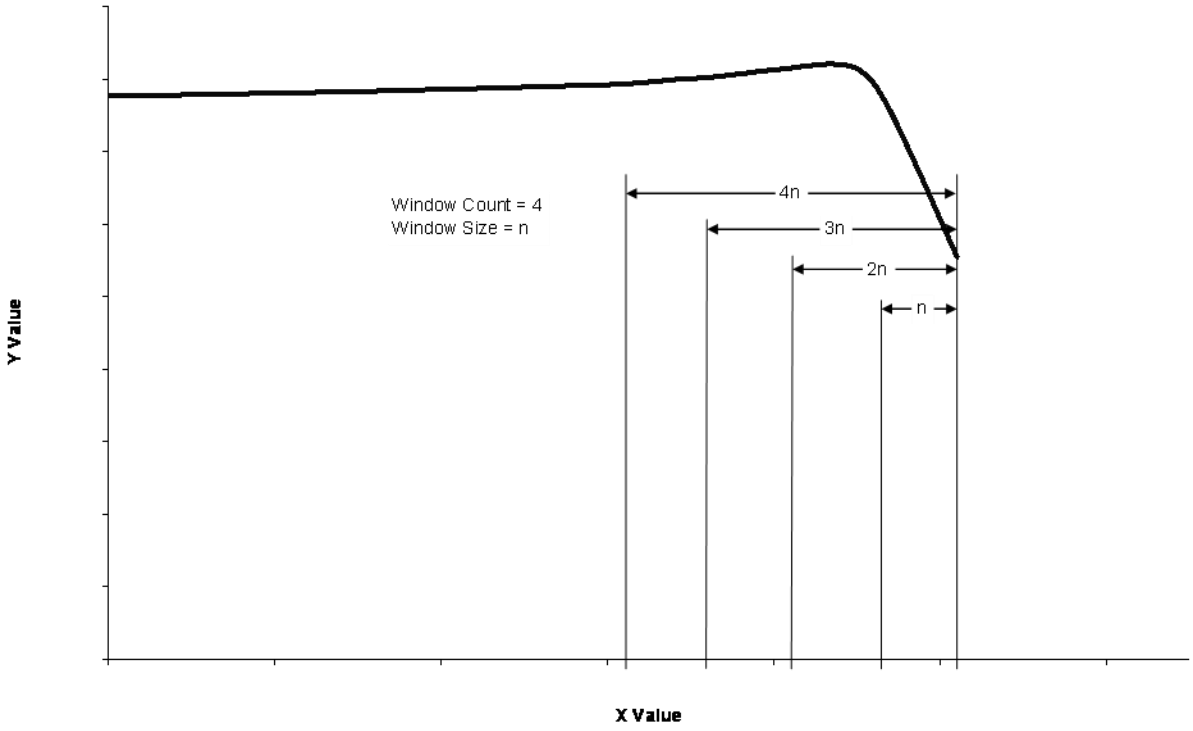

Figure 3-2 Window Size and Window Count are two governing parameters for the Statitics-Based N^{th} Order Polynomial Predictor. Here the relationship between the dataset, Window Size and Window Count is shown.

CHAPTER 4
IMPLEMENTATION DETAILS

In this chapter, the author outlines the technical details of a reference implementation of the architecture and prediction algorithm presented in the previous chapter. Many technical challenges existed in implementing the theoretical Spatiotemporal World Model and the Statistics Based Nth-Order Polynomial Predictor. Discussion of those challenges and the way each was overcome is detailed herein. The JAUS World Model Vector Knowledge Store Message Set section discusses the details of the JAUS messages implemented to embed the theoretical architecture into the research environment available. The Database Selection and Design section discusses the geospatial database selected for use in this implementation and the design of the database's tables. To easily and flexibly implement a variety of prediction techniques, a Generic Predictor interface was designed and is described in the section titled the same. The Polynomial Predictor section outlines the details of the reference implementation of the SNOPP algorithm. Lastly, the World Model Vector Knowledge Store section details how the various pieces were brought together into a single JAUS component.

JAUS World Model Vector Knowledge Store Message Set

The Center for Intelligent Machines and Robotics (CIMAR) has been involved in the activities of the Joint Architecture for Unmanned Systems (JAUS) Working Group for several years through sponsored work with the Air Force Research Lab (AFRL) at Tyndall Air Force Base. As such, the bulk of the research work done at CIMAR focuses on the usability and extension of JAUS to solve a large variety of ever-increasingly complex robotic system problems. As discussed in Chapter 2, recent work within the JAUS Working Group has focused on the area of world modeling. However, the work done prior to the author's research was primarily focused on the query and storage of static geospatial data objects. Therefore the

research presented herein was focused on extending the existing World Model Vector Knowledge Store (WMVKS) component to support the generic Spatiotemporal World Model Architecture defined in the previous chapter.

This approach provided a number of key advantages. Firstly, a vast and reliable collection of software which has been developed at CIMAR to aid in the development and deployment of JAUS components. Secondly, the ability to quickly insert the author's experimental component into an existing JAUS robotic platform for field testing and evaluation. Lastly, a wealth of knowledge about the JAUS architecture and its capabilities allowed the author to focus more on application details and less on basic infrastructure of the system.

The JAUS WMVKS message set is based upon three primary entities; the object geometry, feature classes and feature class attributes. The object geometry can be one of a point, line or polygon along with the associated global coordinates of each vertex of the geometry. The current WMVKS message set only supports the global latitude and longitude coordinate system based on the WGS84 ellipsoid. This is the primary global coordinate system defined for use in JAUS. Associated with each object is also one or more feature classes. Feature classes are used to categorize the geometries. Examples of feature classes may include roads, terrain, occupancy, trees, etc. The feature classes can be closely associated with different layers of information in typical geospatial information (GIS) system. Associated with each feature class is a single feature class attribute. A feature class attribute can be of several different types (byte, integer, float, RGB, etc). These attributes provide more detailed data about the feature class they describe. For example, an object may have the "Velocity_Meters_Per_Second" feature class associated with it, and its attribute value may be "11.2." This value would indicate the velocity of the object it is attached to. In the reference implementation, both objects and their feature class

attributes can be considered dynamic by the WMVKS. The message set supports the ability to designate separate estimation algorithms for each object and each feature class attribute. This allows a large variety of objects and feature classes to be handled by the component.

In order to successfully deploy the WMVKS as a Spatiotemporal World Model, the existing WMVKS message set had to be modified and extended to support a large variety of new data. Table 4-1 lists the various messages and implemented in the WMVKS component developed for this research. Messages which did not exist in the original WMVKS message set and have been added for this research are indicated in the appropriate column. The message set definition is included as Appendix A.

Some key messages and their content were developed specifically to support the author's work. These include the Query Vector Knowledge Store Objects Future State, Report Vector Knowledge Store Objects Future State, and Modify Vector Knowledge Store Objects messages. The original WMVKS message set did not provide an interface to allow a client to change or modify an object's state after insertion. The new modify message addresses this need. The original WMVKS message set also did not address the ability to query about time-variant data. To that end, the query and report future state messages were developed. These messages extend the functionality of the original knowledge store, adding the ability to reason about the future of an object or its attributes.

Database Selection and Design

A significant challenge of world modeling task was to implement some methodology to store the knowledge store's objects and attributes. As discussed in Chapter 2, Spatial Databases are relational or object-relational database management systems (DBMS) which specialize in the handling of spatial data. The DBMS approach was chosen for the author's work because it provided a number of advantages such as

- persistent storage capabilities,
- high-level (e.g. SQL) interface languages,
- availability of robust and complete spatial query capabilities,
- support of temporal data types and queries, and
- portability of code between projects.

A constraint of the development environment at CIMAR is that all the robotic systems run on some version of the Linux operating system. This is due to a number of factors which are themselves outside the scope of this discussion. However, in the selection of a candidate spatial database technology, it was a requirement for the purposes of this work. The author was also interested in attempting to accomplish the goals of the work through the use of an open-source implementation. While various DBMS technologies are available cheaply or for free to educational institutions, a majority of the author's work was destined for use by AFRL, where the DBMS of choice could cost significantly more. See Table 2-1 for more details about various Spatial DBMS technologies.

PostgreSQL is an open source object-relational DBMS. PostGIS is a collection of data types and functions which "spatially extend" PostgreSQL to allow the database to function as a spatial knowledge store. PostGIS is also available open source. A companion software library, Geometry Engine Open Source (GEOS) [48], is a C/C++ API which implements the OpenGIS Consortium's Simple Features Specification for SQL [29]. This is important because the bulk of the work done at CIMAR is done in the C/C++ programming languages. PostgreSQL, PostGIS and the GEOS library provide the core geospatial capabilities of the implementation. PostgreSQL also supports data types for temporal data. This allows support for the storage and querying of time stamps in the database, providing the necessary temporal capabilities for the implementation.

The design of a database schema was also required. The goal was to easily and efficiently store a large variety of objects within the database which could be quickly and easily queried. Two tables were created to store the objects and their related feature classes. Figure 4-1 shows the Entity-Relationship Diagram for the database design and the various field names for each table.

The Objects table consists of 9 fields: object_id, object_type, buffer_meters, object_timestamp, utm_init_longitude, estimator_type, estimator_solution, vector_object, and geometry. Table 4-2 lists these fields, their PostgreSQL data type and a brief description of each. The FeatureClass table consists of 7 fields: object_id, fc_id, fc_estimator_type, fc_estimator_solution, fc_data_type, attribute and attribute_timestamp. Table 4-3 lists these fields, their PostgreSQL data types and a brief description of each. To ease in the use of these two tables, a unique SQL VIEW is constructed in the database. This is constructed as a SQL JOIN of the two tables with the following statement:

```
CREATE VIEW snapshot AS
SELECT * FROM objects JOIN featureclasses
USING (object_id);
```

This allows quick and easy access to the combination of data from the two tables and is used extensively in the various query responses.

Generic Predictor Interface

One key element to the overall Spatiotemporal World Model Knowledge Store Architecture is the ability to easily deploy a variety of prediction algorithms. It was important that the reference implementation address this in a flexible and robust manner. To that end, the Generic Predictor Interface was developed. The Generic Predictor provides a common set of functionality that all predictors must implement. The methods supplied by the interface include solver, estimator, toBuffer and fromBuffer. The interface also provides common storage for the

data set that the prediction algorithm is concerned with. It also provides a storage container for any specific parameters that a specific predictor needs. One common configuration parameter is included in the interface. This is a value for the minimum number of data points the predictor can solve for. This is important because some prediction algorithms are unable to find a solution if less than the minimum number of data points are provided. (e.g. a linear solution is not viable given a single data point)

The solver method is used to generically involve a specific algorithm's unique solution. The estimator method similarly provides a common method of retrieving the value of a given predictor for a given data point. Lastly, the toBuffer and fromBuffer methods provide a mechanism by which a specific predictor can pack its critical data into and from a byte buffer. This is used in conjunction with the estimator_solution fields in the database to provide fast, efficient access to the predictor's solution for future state queries. Three different prediction techniques were implemented to support the author's work: a static predictor, a linear predictor and a polynomial predictor. The implementation details of the polynomial predictor are covered in the next section. Details of the static and linear predictors are covered in the Dynamic World Model Vector Knowledge Store section later in this chapter. Excerpts of the Generic Predictor code are provided in Appendix B.

Polynomial Predictor Implementation

The Polynomial Predictor is an implementation of the Statistics-Based N^{th} Order Polynomial Predictor (SNOPP) described theoretically in Chapter 3. One key feature of the polynomial predictor is the incorporation of a number of configuration parameters that control its behavior. These configuration parameters are listed in Table 4-4. The Window Count and Window Size parameters align with the ones outlined in the discussion of SNOPP in the previous chapter. The Minimum and Maximum Order values provide limits on the solution provided by

the predictor. The P-Critical Value parameter is used to control which level of confidence is imposed on the alternative hypothesis test. The Minimum Point Count parameter is inherited from the Generic Predictor and indicates the minimum number of points needed to evaluate the predictor. This is usually driven by some combination of Window Size and/or Maximum Order.

The predictor makes use of the GNU Science Library (GSL) [49] for its multivariable least-squares fit functions. The `gsl_multifit_linear()` method fits a function of the form $\hat{\boldsymbol{y}} = \mathcal{X}\hat{\boldsymbol{\beta}}$. The $\hat{\boldsymbol{y}}$ vector is formed from the collection of values from the object or attribute being predicted. Each row of the $\mathcal{X}$ matrix is the corresponding timestamp value, raised to the powers 0 through *k*. This casts the multivariable least-squares regression in the form of a k^{th} order polynomial, i.e. $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_n x^k$.

This predictor also makes use of GSL to evaluate the F-Value obtained from equation (3.8). To evaluate this, the cumulative distribution value Q(x) is found using the `gsl_cdf_fdist_Q()` function. This value is evaluated against the P-Critical Value provided to analyze the alternative hypothesis outlined in Chapter 3. This yields the statistically significant order solution (*p*). For each window (defined by Window Count) the Polynomial Predictor compares the output solution for $\hat{\boldsymbol{\beta}}$. As outlined in Chapter 3, the solution with the lowest order and greatest window size is chosen.

To aid in the ease of use by the WMVKS, the Polynomial Predictor implements the Generic Predictor Interface discussed previously. To do so it must provide functions for the solver, estimator, toBuffer and fromBuffer methods. The solver and estimator functions are straight-forward and either solve for the prediction equation, or estimate using that solution. The toBuffer method packs a collection of solution-critical values into a byte buffer. These include

the solution's order and the $\widehat{\boldsymbol{\beta}}$ vector. The fromBuffer method does the inverse operation. This provides a quick and flexible way to store the vital information for future state queries.

Dynamic World Model Vector Knowledge Store

The purpose of this chapter, thus far, has been to outline the building blocks that make up the World Model Vector Knowledge Store (WMVKS) that was implemented to support the author's work. Figure 4-2 shows an outline of the WMVKS as implemented. The WMVKS was implemented as a JAUS Component based on a common component template developed at CIMAR for previous JAUS work. This allowed the author to focus efforts on the implementation of the WMVKS behavior and not the overall JAUS architectural requirements. Each message outlined in Table 4-1 needed to be implemented to be used by the WMVKS. The code to implement the various messages is also based on a flexible skeleton created previously at CIMAR for JAUS messages.

The WMVKS is an event-driven component. That is, it only has to process data as requests and queries are received and has no observed periodic behavior (outside the normal JAUS periodic behavior which is a 1 Hz Report Heartbeat Pulse message). Therefore, its state machine is rather simple. JAUS defines several valid states of a given component. However, the WMVKS typically transitions straight to READY unless some error or fault is detected. When a message is received, the component does whatever work is necessary to process and answer that request.

As discussed previously, a key feature of the overall Dynamic World Model Knowledge Store architecture is the ability to deploy different prediction algorithms. To support the author's work, three unique prediction algorithms were implemented; a static predictor, a linear predictor and the polynomial predictor discussed previously. Each predictor implements the Generic Predictor interface to allow ease of use by the WMVKS. The static predictor is the most simple.

It is used as the prediction algorithm for any static object or attribute and is included for completeness in the design. It stores the latest value from the dataset as the solution and returns this as the estimated value for any point in time. The linear predictor implements a simple least-squares linear fit to a set of data. For the implementation used, up to ten data points are used to fit the data. The linear fit predictor stores its result in a similar manor to the polynomial predictor through the use of the toBuffer and fromBuffer methods of the Generic Predictor interface. The least-squares approximation is done using the `gsl_fit_linear()` method from the GSL library.

The static and linear prediction algorithms were implemented to provide robust solution future state queries. While the polynomial predictor was the primary prediction method developed and tested for this dissertation, it was realized that no valid solution would exist for data sets prior to the minimum point count defined in the polynomial predictor. Rather than handle this as a special case in that predictor, the author made use of the flexible predictor capabilities to implement lower count prediction techniques. Therefore, when a data set is made up of a single value, the static predictor is automatically invoked, regardless of the estimation type identified (the original estimation type value is preserved and left unmodified in the database record). If more than a single data point is present, but less than the specified predictor's minimum point count, the linear predictor is used. This not only allowed a more robust solution to state prediction, it also increased the capability of the WMVKS by providing three prediction algorithms instead of just the one.

One of the more complex messages handled by the WMVKS is the Create Vector Knowledge Store Object(s) message. This message may contain one or more objects with various attributes which must be handled by the WMVKS and added to its current collection of

objects. The code for this procedure is included as part of Appendix B. The following is a step-by-step explanation of this process:

1. The received Create Vector Knowledge Store Object(s) message is "unpacked" from the byte buffer into its specific data structure.
2. For each object, the object id bit is checked to see if it has been included, if not, the next valid value is queried from the database.
3. Each object is translated from its data structure into an SQL INSERT statement for storage into the Objects table of the database:
```
INSERT INTO objects(<table fields>)
VALUES (<message data>);
```
4. Each feature class included with an object is similarly converted into an SQL INSERT statement for insertion into the FeatureClass table of the database.
```
INSERT INTO featureclass(<table fields>)
VALUES (<message data>);
```
5. If the confirmation has been requested, a Report Vector Knowledge Store Objects Creation message is sent back to the originating client with a list of the object ids which have been stored.

Similar to the create message, the Modify Vector Knowledge Store Object message invokes a complex process in the WMVKS. This process involves the various prediction algorithms; invoking them to provide solutions for future state estimation. Each time an object is modified, new information is included in the state history. This allows a new, updated solution to be found for future state estimation. The more often an object's state is updated, the more quickly the various predictors may be able react to changes in the observed behavior and account for it in the future state estimator. Using the prediction methods implemented for the author's work, prediction of an object's position is limited to estimating the centroid of the object. This means the bounds of a polygon or line are considered static for the author's work. This assumption may not always be true. Since the prediction algorithms implemented in the WMVKS are only concerned with the state of a single time-varying dataset, the estimation of future geospatial position is broken down into two datasets (Easting vs. Time and Northing vs. Time). The position solution is then the combination of two separate prediction solutions. Position estimation is done in the Universal Transverse Mercator (UTM) coordinate system

rather than the Latitude, Longitude and Altitude (LLA) coordinate system so that Euclidean Geometry can be applied. Projection between coordinate systems is provided by a UTM library that is part of CIMAR's core library set.

The code to handle receipt of a Modify Vector Knowledge Store Objects message is also included in Appendix B, but a step-by-step account is provided here for clarity:

1. The received Modify Vector Knowledge Store Objects message is "unpacked" from the byte buffer into its specific data structure.
2. The object id included in the received message is checked against the database. If the target object does not exist, an error may be reported.
3. The vector_object data field is retrieved from the database through the use of a SQL SELECT statement and used to create a copy of the database object in memory:
```
SELECT vector_object FROM objects
WHERE object_id = <message_object_id>;
```
4. The information contained in the modify message is compared to the information retrieved from the database. Certain fields (such as buffer_meters) are replaced if modified. Modification of values in the database is done using a SQL UPDATE statement:
```
UPDATE objects SET buffer_meters = <value>
WHERE object_id = <message_object_id>;
```
5. In the case of a dynamic object or attribute, the WMVKS adds the included information to the existing object or attribute history, as appropriate. If the value being modified is considered static, the modification message is used to replace the current value with the modified one.
6. The modified and/or updated version of the vector_object information is generated and updated in the appropriate field of the database.
7. Since object information has changed, the prediction solution must be updated. For each feature class, the appropriate predictor (static, linear or polynomial) is called and the solution is stored in the fc_estimator_solution field of the FeatureClass table. For every object, the estimator_solution is populated with the prediction solution for both the Easting (X) and Northing (Y) values.
8. If the confirmation bit has been requested, the WMVKS responds to the originating client with a Report Vector Knowledge Store Object Modification message.

The create and modify messages provide the primary methodology for a data provider (such as a sensor) to insert or change data within the WMVKS. However, other clients may be primarily interested in the consumption of that information. Clients interested only in the current state of an object or some attribute can use the Query Vector Knowledge Store Object(s) message. This message retrieves the object from the database through a SQL SELECT statement.

This query message is flexible in the way it can be constructed, from boundary queries to specific objects and/or attributes

The other approach that can be employed is to query the future state of an object or attribute. This is done through the use of the Query Vector Knowledge Store Object(s) Future State message. This message can be used in two ways, designated by the query type field. In the he first, called Absolute Query, the timestamp provided is considered an absolute time in the future at which to query the state of an object and its attributes. The other, Relative Query, is used to specify some time offset which will be added to the current time at which the WMVKS receives the message. This feature was added to the message to ease the use of registered periodic events (called Service Connections in JAUS, this mechanic is outside the scope of the author's discussion [1]). In either case, the query time is used by the appropriate predictor to generate an estimate for the future value of the object and its attributes. The code for handling a Query Vector Knowledge Store Object(s) Future State message is also included in Appendix B. An outline of this code is provided below:

1. The received Query Vector Knowledge Store Object(s) Future State message is "unpacked" from the byte buffer into its own data structure.
2. For each object specified in the query, the database is checked to ensure an object with that id value exists.
3. For each object queried, the query time value is setup, either using the absolute or relative time method. Note, this means multiple future state queries can be setup with different query times. The advantage of this is the ability to construct a query for the future state of a single object at multiple query times or offsets using a single message.
4. For each object queried, the appropriate object predictor solution (in byte buffer format) is read from the database using an SQL SELECT statement:

```
SELECT estimator_solution FROM objects
WHERE object_id = <message_object_id>;
```

5. This object predictor solution is unpacked appropriately (into both an X and Y predictor) and then used to estimate the position of the object at the given query time using the estimate function.
6. For each object queried, the appropriately queried attributes are also estimated. To do so, the attribute estimator solutions are first retrieved from the database using a SQL SELECT statement. They are then converted using the predictor's fromBuffer method.

Lastly, state estimates are conducted using the appropriate query time and estimator method.

7. The object's position estimation and any appropriate attribute estimations are then returned to the querying client via a Report Vector Knowledge Store Object(s) Future State message.

The WMVKS component provides a robust implementation of the Spatiotemporal World Model architecture outlined in Chapter 3. It does so by combining the capabilities of the JAUS World Model Vector Knowledge Store message set, the advantages of the PostgreSQL database, and the capabilities of an array of unique prediction techniques. The techniques and approach outlined above provide a unique solution to the problem of storing and querying the future state of a dynamic object or attribute within the JAUS architecture. Many of the same techniques could easily be applied to any other robotics architecture, provided the messaging interface could be modified to support the message interface introduced in Chapter 3.

Table 4-1 JAUS WMVKS Message Set as implemented in the reference World Model Vector Knowledge Store component developed to support the author's research. Both new and extended JAUS messages are listed.

Message Name	JAUS Command Code	New Message	Input / Output Message
Request Vector Knowledge Store Object Id Block	F120h	●	Input
Assign Vector Knowledge Store Object Id Block	F320h	●	Output
Create Vector Knowledge Store Objects	F121h		Input
Report Vector Knowledge Store Objects Creation	F321h		Output
Modify Vector Knowledge Store Objects	F122h	●	Input
Report Vector Knowledge Store Object Modification	F322h	●	Output
Delete Vector Knowledge Store Objects	F123h		Input
Report Vector Knowledge Store Objects Deletion	F323h	●	Output
Query Vector Knowledge Store Objects	F130h		Input
Report Vector Knowledge Store Objects	F330h		Output
Query Vector Knowledge Store Geospatial Bounds	F132h		Input
Report Vector Knowledge Store Geospatial Bounds	F332h		Output
Query Vector Knowledge Store Temporal Bounds	F133h	●	Input
Report Vector Knowledge Store Temporal Bounds	F333h	●	Output
Query Vector Knowledge Store Objects Future State	F134h	●	Input
Report Vector Knowledge Store Objects Future State	F334h	●	Output

Table 4-2 Object table data fields, PostgreSQL type and description.

Field Name	PostgreSQL Type	Description
object_id	INT4 PRIMARY KEY	Unique object id value assigned to each object.
object_type	INT2	Value of the JAUS WMVKS object type (i.e. point, line or polygon).
buffer_meters	FLOAT	Size of an optional buffer around the object's geometry.
object_timestamp	TIMESTAMP	Last time stamp value associated with the object geometry. Used for temporal queries.
utm_init_longitude	DOUBLE	Value, in radians, of the longitude of the first point

	PRECISION	associated with this object. This is provided to ensure proper projection between LLA and UTM for an object.
estimator_type	INT2	Value of the estimator type associated with this geometry.
estimator_solution	BYTEA	Hex string of the binary array of the estimator solution. Used to quickly provide solutions to future state queries.
vector_object	BYTEA	Hex string of the binary array of the JAUS Vector Object structure.
Geometry	GEOMETRY	PostGIS geometry column used to store the latest geometry value of the object. Used for geospatial queries.

Table 4-3 Feature Class table data fields, PostgreSQL types, and description.

Field Name	PostgreSQL Type	Description
object_id	INT4 PRIMARY KEY	Unique object id value assigned to each object.
fc_id	INT4 PRIMARY KEY	Feature Class id value, non-unique. However, there can only be one fc_id value per object_id value.
fc_estimator_type	INT2	Value of the estimator type associated with this geometry.
fc_estimator_solution	BYTEA	Hex string of the binary array of the estimator solution. Used to quickly provide solutions to future state queries.
fc_data_type	INT2	Value of the enumerated data type of this feature class' attribute.
Attribute	INT8	Feature Class Attribute value, stored as an 8-byte integer.
attribute_timestamp	TIMESTAMP	Last time stamp value associated with the object geometry. Used for temporal queries.

Table 4-4 Polynomial Predictor configuration parameters

Parameter Name	Description
Minimum Point Count	The minimum number of data points for which the predictor can successfully be used. This is usually dependent on the Minimum / Maximum Order parameters and/or the Window Size parameter.

Minimum Order	This is the minimum order polynomial the predictor will analyze for a solution. Usually 0 or 1.
Maximum Order	This is the maximum order polynomial the predictor will analyze for a solution. Higher order polynomials can give better results in dynamic situations, and lower order ones are more stable in extrapolation.
Window Size	This is the size of each data window to analyze.
Window Count	This is the number of data windows, each Window Size larger than the previous, to analyze.
P-Critical Value	This value is used to evaluate the statistical significance of a particular order.

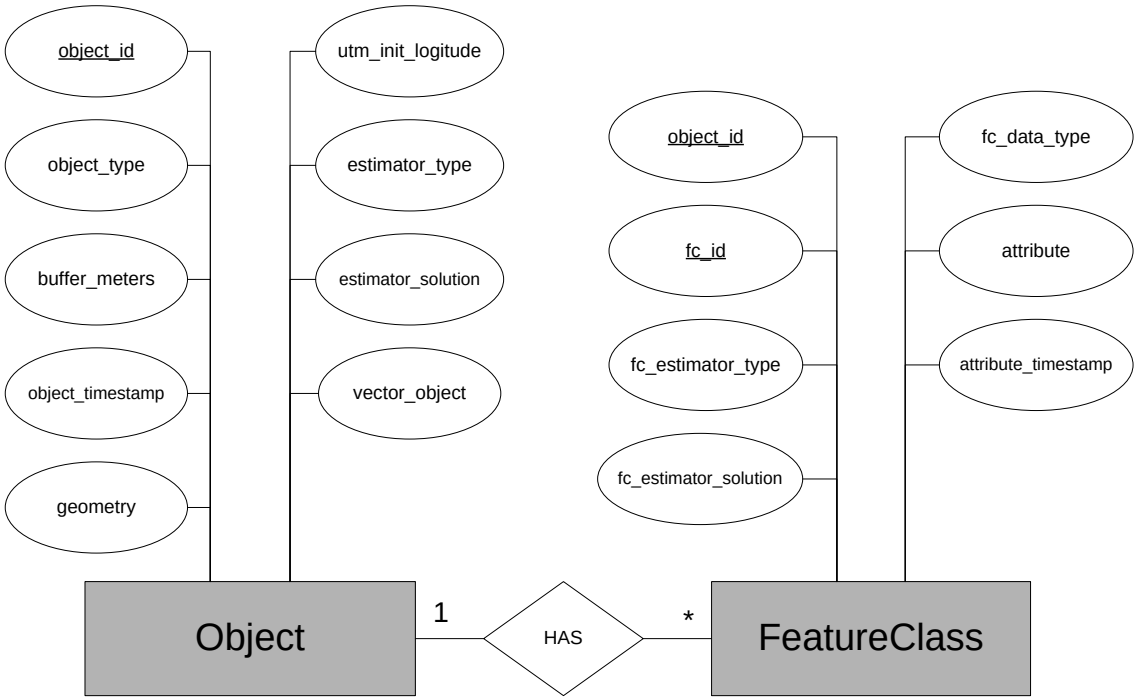

Figure 4-1 Entity-Relationship Diagram of WMVKS Database

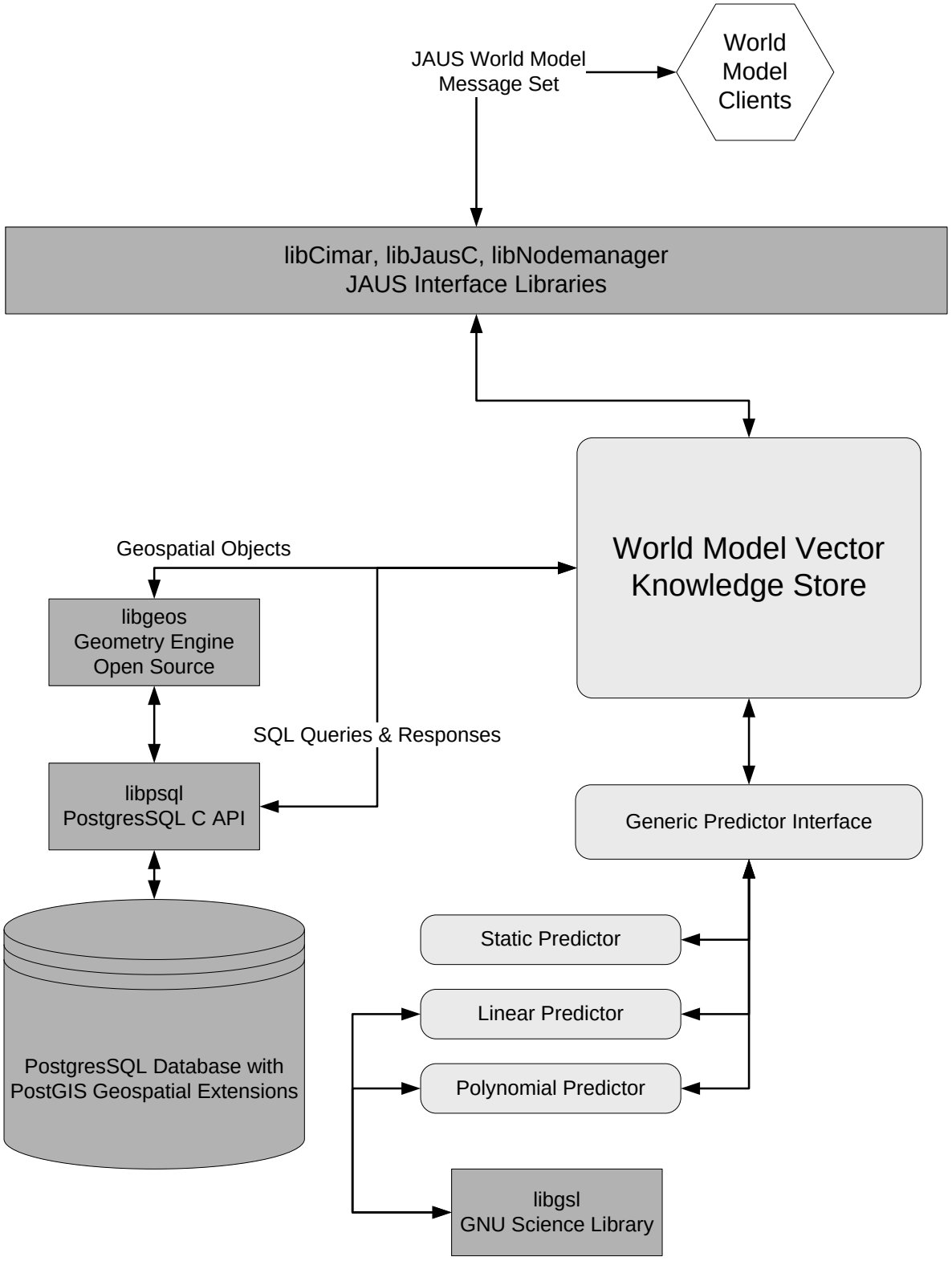

Figure 4-2 World Model Vector Knowledge Store Implementation Diagram.

CHAPTER 5
TESTING AND RESULTS

Previous chapters have dealt with the background and implementation details of the author's dissertation. Herein, the testing methodology and results for a series of test are presented. To test the World Model Vector Knowledge Store (WMVKS) component a component capable of behaving as a source of dynamic information was required. The Laser Tracking Smart Sensor section covers a component developed to address this need. The Test Procedure and Plans chapter describes the various test scenarios devised and how each was conducted. Lastly, the Test Results section provides an overview of the results obtained by the author's work.

Laser Tracking Smart Sensor

In order to properly test the WMVKS and its implementation, a source of dynamic information was needed. For the author's work, a simple laser-based object tracking algorithm was implemented to collect real-world test data which could used to test the knowledge store. The Laser Tracking Smart Sensor (LTSS) was developed based on a combination of previous research activities at CIMAR.

In the spring of 2003, a laser-based tracking algorithm was developed by the author and another researcher at CIMAR to support the "follow-the-leader" challenge at the Intelligent Ground Vehicle Competition (IGVC). This challenge was to follow at some fixed distance a human-driven lawn tractor. However, the algorithm has shown effectiveness at tracking and following a number of objects including other robots, humans and larger vehicles.

The laser-tracking algorithm works by first receiving a user input which "seeds" the system and identifies the object to be tracked (Figure 5-1a). The system then searches for all neighboring points within some threshold distance to the object. Once all neighboring points are

identified, a bounding box around the collection is established (Figure 5-1b). On successive laser scans, the system uses an estimated bounding box to seed itself and repeat the threshold based search. The system is controlled to allow the collection of points to grow and shrink at some user-defined rate.

The algorithm builds up a history of the tracked object's position and uses them to estimate the velocity of the object. This velocity is in turn used to estimate the future position of the object at successive laser scans. Due to the fast (35-70Hz) rate at which laser scans occur, a linear velocity prediction has yielded adequate results. The LTSS algorithm has been shown to be incredibly capable at not only tracking a visible dynamic object, but estimating and reacquiring the object if it travels through the shadow of another object. Figure 5-2 shows the LTSS algorithm in action. The selected object is shown in orange while the current and future bounding boxes are shown in blue and green respectively. The estimated velocity vector is also shown.

This algorithm successfully enabled the CIMAR entry to complete the "follow-the-leader" course in the first attempt at the 2003 IGVC competition. CIMAR's entry (the TailGator) was the only vehicle to complete the course at the competition. Figure 5-3 shows the TailGator platform on the course following a human-driven lawn tractor.

More recent work has been done by the author to integrate this laser algorithm into the JAUS system architecture onboard the NaviGATOR. For the DARPA Grand Challenge contest, CIMAR developed a common sensor architecture called Smart Sensors. The LTSS is based on this architecture which allows easy interfacing with the existing JAUS infrastructure onboard the NaviGATOR. More information about the Smart Sensor architecture can be found in [50].

The IGVC laser algorithm was integrated into a smart sensor and interfaced to the planar laser mounted on the front of the NaviGATOR. Figure 5-4 shows the output of this Smart Sensor and provides a brief description. This setup was used track a truck driving in front of the NaviGATOR as a test location near the university. These tests provided a collection of data used to evaluate the performance of the WMVKS. Figure 5-5 shows the test setup with the NaviGATOR and the truck used as the tracked object.

Test Procedure and Plans

The LTSS data was used to test the WMVKS and its associated algorithms. For the author's work, log files were generated by the LTSS which contained the tracked object's position and a timestamp for each iteration of the LTSS. Also logged at the same time were the NaviGATOR's position, velocity and heading values. This provided a variety of time-variant data which could be used to evaluate the performance of the WMVKS. Two test setups were devised. In the first test, the NaviGATOR was stationary while a moving object passed in front of it. This object was detected, then tracked by the LTSS and its position logged. Since the NaviGATOR is stationary for this test, it position, velocity and heading data is not used in the author's work. In the second test (shown in Figure 5-5), the NaviGATOR is driven some distance behind a large truck in a looped course. The truck is detected and tracked throughout the loop. Once again the object's position is logged, along with the NaviGATOR's position, velocity and heading. Each test was performed five times. The log files were later taken and played-back in real time to the WMVKS and the results of the prediction algorithms recorded. To test the capabilities of the WMVKS, these log files were used to analyze five different test scenarios. Each scenario is described below along with a respective test plan.

Test Scenario 1: Tracked Object Using Polynomial Predictor from Stationary Platform

In this scenario, the tracked object is moving while the NaviGATOR is stationary. It demonstrates basic behavior of the tracking algorithm and WMVKS performance. The polynomial predictor is used to estimate the future position of the tracked object one second, two seconds, three seconds, four seconds and five seconds in the future. Table 5-1 outlines the test plan; included therein is the test purpose, hypothesis, expected results, test design and logged data.

Test Scenario 2: Tracked Object Using Polynomial Predictor from Moving Platform

In this scenario, the tracked object is moving while the NaviGATOR is following it at some distance (both vehicles are human-driven for this test). This test demonstrates performance of the WMVKS for a common scenario, which is tracking objects moving around the vehicle while the vehicle itself is in motion. The polynomial predictor is used to estimate the future position of the tracked object one second, two seconds, three seconds, four seconds and five seconds in the future. Table 5-2 outlines the test plan; included therein is the test purpose, hypothesis, expected results, test design and logged data.

Test Scenario 3: Tracked Object Using Linear Predictor from Moving Platform

In this scenario, the tracked object is moving while the NaviGATOR is following it at some distance (both vehicles are human-driven for this test). This test is done in contrast to scenario two in using the same inputs but analyzing it using the linear predictor instead. The linear predictor is used to estimate the future position of the tracked object one second, two seconds, three seconds, four seconds and five seconds in the future. Table 5-3 outlines the test plan; included therein is the test purpose, hypothesis, expected results, test design and logged data.

Test Scenario 4: Velocity Attribute Using Polynomial Predictor

In this scenario, the WMVKS is used to predict the future state of a feature class attribute rather than an object's position. This shows the ability of the WMVKS to handle data that is not geospatial in nature. The data used in this test was collected from the NaviGATOR's Velocity State Sensor (a standard JAUS component) during the test runs described previously. The polynomial predictor is used to estimate the velocity of the NaviGATOR one second, two seconds, three seconds, four seconds and five seconds in the future. Table 5-4 outlines the test plan; included therein is the test purpose, hypothesis, expected results, test design and logged data.

Test Scenario 5: Heading Attribute Using Polynomial Predictor

In this scenario, the WMVKS is used to predict the future state of a feature class attribute which represents the heading (or orientation) of some object. Data used for this test was collected from the Global Position Sensor component (a standard JAUS component) onboard the NaviGATOR. The polynomial predictor is used to estimate the state of the attribute one second, two seconds, three seconds, four seconds and five seconds in the future. Table 5-5 outlines the test plan; included therein is the test purpose, hypothesis, expected results, test design and logged data.

Test Results

In this section, the data collected for each test scenario will be analyzed and presented. Each test scenario was conducted five times. In this chapter, only one of each of these tests will be presented and discussed. Representative charts for each scenario are presented.

Test Scenario 1 Results

The purpose of this test scenario is to show the position estimation capabilities of the WMVKS as implemented by the author. As discussed in Chapter 4, to estimate the position of an

object, two polynomial predictors are employed, one for the easting (x) value and one for the northing (y) value. The position values are estimated in the Universal Transverse Mercator (UTM) coordinate system and converted into Latitude and Longitude prior to being sent to the LTSS. The test plan for this test is included as Table 5-1.

Figure 5-6 show a typical plot of the easting vs. northing values used as source data for this series of tests. This chart shows that the tracked object moved primarily along the north/south axis with very little movement in the east/west direction. In figure 5-7 easting and northing values are plotted separately with respect to time. Figure 5-8 shows the easting vs. time plot with the one second, three second and five second prediction values. Figure 5-9 shows the northing vs. time plot with one second, three second and five second prediction values. Figure 5-10 shows the combined easting vs. northing plot with the predicted values.

Overall, test scenario 1 has demonstrated the ability of the WMVKS to handle dynamic position data, reason about the future state of that data and provide feedback to the LTSS. It has shown successful use of the polynomial predictor algorithm and the new JAUS world model message set implemented by the author.

Test Scenario 2 Results

The purpose of this test is to demonstrate the ability of the WMVKS to handle prediction of an object tracked by the LTSS over a much longer distance and time span. The test data covers over 3 minutes and 800 meters in length. Figure 5-11 shows a plot of easting vs. northing for a typical data set from this test.

Figure 5-12 shows the same data plotted against time with easting on the left axis and northing on the right axis. Figure 5-13 provides a plot of just the easting value vs. time and the one second, three second and five second prediction values. Due to the scale of this plot, details are difficult to read. Therefore Figures 5-14, 5-15 and 5-16 provide closer views of the more

critical sections of the plot. Figure 5-17 shows the northing vs. time plot with corresponding prediction values. Again, Figures 5-18 and 5-19 are provided for clarity.

Figures 5-20 and 5-21 show, graphically, the error in the 1 second predictor plotted on one axis with the actual and predicted values plotted on the other axis. These charts show that the polynomial predictor demonstrates large errors around changes in the data trend, but quickly reacquires the proper trend and provides good prediction results.

The previous charts show graphically the error in the polynomial predictor. An analysis was conducted which calculated the error in the predicted value for all five trials of this scenario; the results of which are presented in Table 5-6. Here, the performance of the polynomial predictor can be shown to yield favorable results. The average error on the 1 second predictor is 0.68 meters for the easting value and 0.60 meters for the northing values. Combined there is an overall 1.26 meter average error in the predicted position compared with the recorded true position 1 second in the future. The standard deviation value for the 1 second prediction was calculated to be 1.89 meters for the combined total of easting and northing. This means that if one assumes the error is normally distributed, 68% of the data falls within one standard deviation and 95% of the data is within two standard deviations (3.78 meters here) from the average value. Therefore for the 1 second predictor, 68% of the time, the error in the predicted value will be between -0.6 and 3.2 meters.

Test Scenario 3 Results

The purpose of this test is to show the ability of the WMVKS to apply a different prediction algorithm to the same set of input data. Using the same data set used in scenario 3, the position of the tracked object is estimated using the linear predictor.

As the same initial data set is used in both scenario 2 and 3, figures 5-11 and 5-12 again illustrate the source data. Figure 5-22 provides a plot of the easting value vs. time including the

one second, three second and five second prediction values. Due to the scale of this plot, details are difficult to read. Therefore, figures 5-23, 5-24 and 5-25 provide closer views of more critical areas of the plot. Figure 5-26 provides a plot of the northing value vs. time including the one second, three second and five second prediction values. Again, figures 5-27 and 5-28 are provided for clarity.

The plots shown in figures 5-20 through 5-24 demonstrate the prediction qualities of the linear prediction algorithm. Compared to figures 5-13 through 5-19, the linear predictor does not yield results as accurate as the polynomial predictor. The linear predictor does exhibit a slightly better response time to changes in trends, but has a much larger spread in the data at any given time. This is most likely due to the rather small maximum point size used for this test (10). A smaller value provides quicker response to trend changes, but less stable results overall due to less data history in the estimator's solution.

Again, an analysis of the error was conducted and the results are presented in Table 5-7. Figure 5-29 shows the error values with respect to time plotted alongside the actual and predicted values for the 1 second prediction of the easting value. Figure 5-30 shows the same plot for the northing data. The values calculated in the error analysis demonstrate that the polynomial prediction method provides better overall results in this test scenario than the linear predictor. The linear predictor has a 1.55 meter error on average in the easting data (for a 1 second prediction) and a 0.89 meter error in the northing data. This yields a combined average error of 2.44 meters for a 1 second prediction. This value is 1.2 meters more than the polynomial predictor for the same data set and prediction interval which is a 91% difference.

Test Scenario 4 Results

The purpose of this test scenario is to demonstrate the WMVKS' ability to handle dynamic feature class attributes and reason about the future state of those attributes. The velocity of the NaviGATOR over the course of a 3 minute drive is used as the sample data set.

Figure 5-31 shows a plot of the velocity data vs. time which is typical of this test scenario. Figure 5-32 shows the same velocity vs. time plot combined with the predicted values one second, three seconds and five seconds in the future. Figure 5-33 shows a plot of the 1 second predictor's results. Here the error vs. time is plotted alongside the actual and predicted values. A summary of the error results in included in Table 5-8.

This test successfully demonstrates the ability of the WMVKS to predict the future state of dynamic feature class attributes. It also demonstrates the ability to make use of estimation techniques for a variety of data. Here the polynomial predictor is used as the prediction algorithm and yields favorable results with an average error of 0.38 meters per second for the 1 second predictor and a standard deviation of 0.34 meters.

Test Scenario 5 Results

This last test scenario is designed to demonstrate something that was mentioned in Chapter 3 when the SNOPP algorithm was introduced. This prediction algorithm does not make use of any parameters which indicate the nature or model of the data being predicted. Therefore, it cannot adequately handle discontinuous data. Heading data exhibits this discontinuous behavior anytime it crosses the $-\pi / \pi$ threshold. This test demonstrates the behavior of the polynomial prediction algorithm when such a discontinuity is encountered. Figure 5-34 shows a plot of the heading data vs. time. Figure 5-35 shows the same data and included the predicted values for one second, three seconds and five seconds in the future. Figure 5-36 shows the 1 second error plot for the same data set. Error analysis results are included in Table 5-9.

Before and after the discontinuity point (at approximately 133 seconds) the polynomial predictor shows prediction capabilities similar to those observed for other dynamic data sets. However, one can observe that immediately after the discontinuity point the predictor's solution is wildly incorrect. In actuality it predicts values well outside the range of $-\pi$ to π which is considered valid for heading values. The effect of this can be seen in both Figure 5-36 (with error values approaching 12 radians) and Table 5-9 where the average heading error even 1 second in the future is 0.26 radians but the standard deviation is 0.92 radians. Much of the error and size of the standard deviation can be attributed to the large errors encountered around the discontinuity point.

Testing Summary

The test scenarios described in this chapter have highlighted the primary functionality of the WMVKS. The JAUS message set introduced in chapter 4 has been shown sufficient to support a large variety of data types and test scenarios. The ability to predict using different prediction algorithms has also been demonstrated in test scenario 3. Lastly, the polynomial predictor has been shown to be flexible enough to be used for a large variety of data in the prediction of future state. It has yielded favorable results in tracking not only objects, but also attributes as well. The linear prediction method was compared to the polynomial predictor and shown to yield less favorable results overall and in general provide much less stable behavior when used to predict future states.

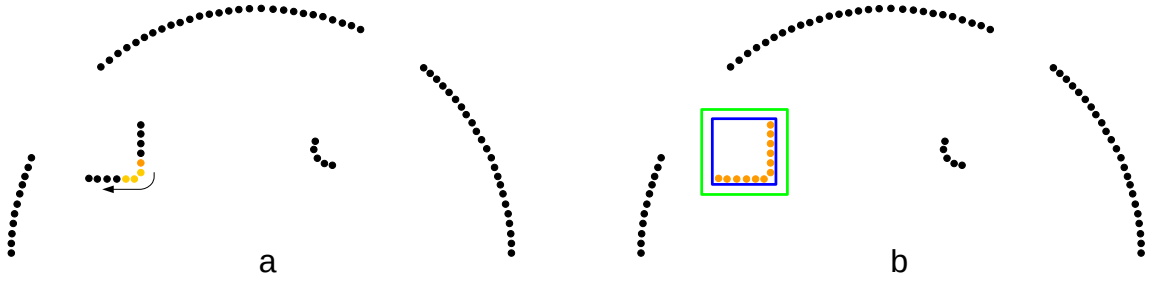

Figure 5-1 Laser tracking algorithm. In (a) the orange dot is the seed point while the algorithm searches and identifies close neighbors in yellow. In (b) the search has been completed and the object to be tracked is surrounded by the inner bounding box. The outer bounding box is the estimated future position of the object which has been grown slightly. At first the object's velocity vector is unknown so the future position and current position are coincident.

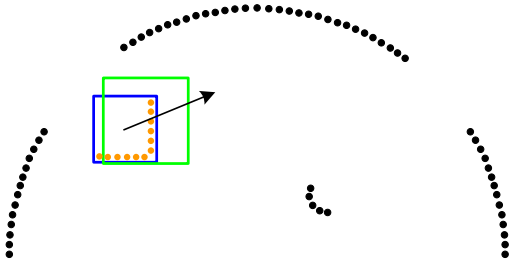

Figure 5-2 Laser tracking algorithm. The tracked object is highlighted in orange. The current bounding box is shown in blue which the estimated future position of the object is shown in green. The estimated velocity vector is also shown.

Figure 5-3 The TailGator using the tracking algorithm to follow a lawn tractor. The TailGator was the only robot at the 2003 Intelligent Ground Vehicle Competition to successfully complete the course.

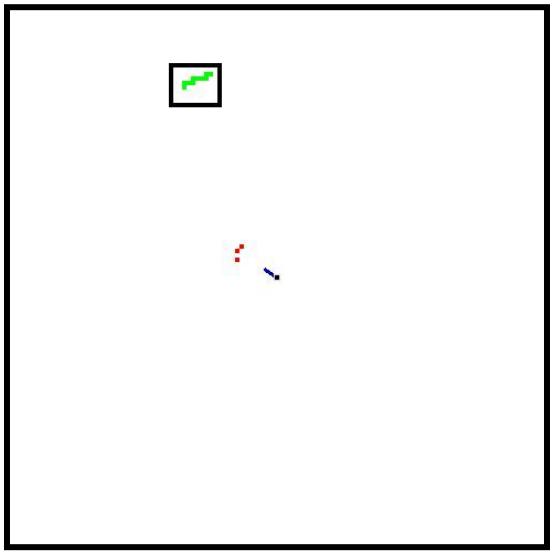

Figure 5-4 Laser Tracking Smart Sensor output. Tracked object shown in green with a black bounding box. Red cells are another non-tracked object. Note the vehicle is located in the center of the grid and the vehicle heading is indicated by the small blue line (north-west here).

Figure 5-5 Laser Tracking Smart Sensor setup for collection of test data. The NaviGATOR (blue) is used to track the moving object (white truck) around a large test circuit.

Table 5-1 Scenario 1 Test Plan

Test Description	In this test, an object detected by the LTSS will be inserted into the WMVKS. The object's position will be updated by the LTSS while its future position will be estimated using the polynomial predictor. For this test, the object will be moving and the robotic platform will be stationary.
Test Purpose	To show the ability of the WMVKS to reason about and estimate the future geospatial position of an object using the polynomial predictor algorithm.
Hypothesis	The polynomial predictor will yield favorable tracking of the object's future position.
Expected Results	The WMVKS will successfully handle a variety of create, modify and query messages to facilitate the creation and monitoring of an object and its future position.
Test Design	For this test, the NaviGATOR is stationary while a truck is driven in front of it. The LTSS identifies and tracks the truck as it moves across the sensor's field of view. The position of the tracked object is updated in the WMVKS at 10 Hz. The future state of the object is queried at 5 Hz. Each time, the position 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future is queried. The following configuration values are used in the polynomial predictor for this test: Minimum Point Count: 10 Minimum Order: 0 Maximum Order: 3 Window Size: 5

	Window Count: 20 P-Critical Value: 0.2 (20%)
Logged Data	The estimated position of the tracked object at 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future with appropriate timestamp for each report.

Table 5-2 Scenario 2 Test Plan

Test Description	In this test, an object detected by the LTSS will be inserted into the WMVKS. The object's position will be updated by the LTSS while its future position will be estimated using the polynomial predictor. For this test, both the object and the robotic platform will be moving. This test is much longer in length than test 1 and will show a larger variety in the data set.
Test Purpose	To show the ability of the WMVKS to reason about and estimate the future geospatial position of an object using the polynomial predictor algorithm.
Hypothesis	The polynomial predictor will yield favorable tracking of the object's future position.
Expected Results	The WMVKS will successfully handle a variety of create, modify and query messages to facilitate the creation and monitoring of an object and its future position.
Test Design	For this test, the NaviGATOR is following a truck which driven in front of it. The LTSS identifies and tracks the truck as it moves. The position of the tracked object is updated in the WMVKS at 10 Hz. The future state of the object is queried at 5 Hz. Each time, the position 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future is queried. The following configuration values are used in the polynomial predictor for this test: Minimum Point Count: 10 Minimum Order: 0 Maximum Order: 3 Window Size: 5 Window Count: 20 P-Critical Value: 0.2 (20%)
Logged Data	The estimated position of the tracked object at 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future with appropriate timestamp for each report.

Table 5-3 Scenario 3 Test Plan

Test Description	In this test, an object detected by the LTSS will be inserted into the WMVKS. The object's position will be updated by the LTSS while its future position will be estimated using the linear predictor. For this test, both the object and the robotic platform will be moving.
Test Purpose	To contrast the difference in results obtained using the polynomial and linear predictors
Hypothesis	The linear predictor will show less optimal prediction capabilities, but more quickly adjust to trend changes.
Expected Results	The linear prediction algorithm will show less optimal prediction capabilities, but perform better than the polynomial predictor around trend changes.
Test Design	For this test, the NaviGATOR is following a truck which driven in front of it. The LTSS identifies and tracks the truck as it moves. The position of the tracked object is updated in the WMVKS at 10 Hz. The future state of the object is queried at 5 Hz. Each time, the position 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future is queried. The following configuration values are used in the linear predictor for this test: Minimum Point Count: 2 Maximum Point Count: 10
Logged Data	The estimated position of the tracked object at 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future with appropriate timestamp for each report.

Table 5-4 Scenario 4 Test Plan

Test Description	In this test, the velocity of the NaviGATOR as it drives some path will be attached to a point object as a dynamic feature class attribute. Its value will be updated and its future value will be predicted using the polynomial predictor.
Test Purpose	To show the ability of the WMVKS to reason about and estimate values for dynamic feature class attributes.
Hypothesis	The polynomial predictor will yield favorable tracking of the velocity value.
Expected Results	The WMVKS will successfully handle a variety of create, modify and query messages to facilitate the creation and monitoring of a feature class attribute and its future state.

Test Design	For this test, the NaviGATOR's velocity value is updated in the WMVKS at 10 Hz. The future state of the value is queried at 5 Hz. Each time, the position 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future is queried. The following configuration values are used in the polynomial predictor for this test: Minimum Point Count: 10 Minimum Order: 0 Maximum Order: 3 Window Size: 5 Window Count: 20 P-Critical Value: 0.2 (20%)
Logged Data	The estimated position of the tracked object at 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future with appropriate timestamp for each report.

Table 5-5 Scenario 5 Test Plan

Test Description	In this test, the heading of the NaviGATOR as it drives will be attached to a point object as a dynamic feature class attribute. Its value will be updated and its future state will be predicted using the polynomial predictor.
Test Purpose	To show the behavior of the polynomial predictor when estimating the value of a non-continuous signal.
Hypothesis	The polynomial predictor will exhibit poor tracking for discontinuous signals.
Expected Results	The polynomial predictor will reasonably predict the heading value between discontinuity points. However at the points of discontinuity the predictor will have very large errors.
Test Design	For this test, the NaviGATOR's heading value is updated in the WMVKS at 10 Hz. The future state of the value is queried at 5 Hz. Each time, the position 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future is queried. The following configuration values are used in the polynomial predictor for this test: Minimum Point Count: 10 Minimum Order: 0 Maximum Order: 3 Window Size: 5

	Window Count: 20 P-Critical Value: 0.2 (20%)
Logged Data	The estimated position of the tracked object at 1 second, 2 seconds, 3 seconds, 4 seconds and 5 seconds in the future with appropriate timestamp for each report.

Table 5-6 Error analysis of test scenario 2

		Easting (meters)	Northing (meters)	Combined (meters)
1 Second	Average	0.676	0.600	1.276
	Standard Deviation (σ)	0.967	1.372	1.892
	2*σ	1.935	2.744	3.784
2 Seconds	Average	1.423	1.236	2.659
	Standard Deviation (σ)	2.071	2.740	4.011
	2*σ	4.141	5.480	8.022
3 Seconds	Average	2.479	2.102	4.581
	Standard Deviation (σ)	3.682	4.556	6.924
	2*σ	7.365	9.111	13.849
4 Seconds	Average	3.856	3.171	7.027
	Standard Deviation (σ)	5.834	6.761	10.561
	2*σ	11.668	13.523	21.122
5 Seconds	Average	5.516	4.454	9.971
	Standard Deviation (σ)	8.495	9.351	14.890
	2*σ	16.989	18.702	29.781

Table 5-7 Error analysis of test scenario 3

		Easting (meters)	Northing (meters)	Combined (meters)
1 Second	Average	1.553	0.891	2.444
	Standard Deviation (σ)	1.472	1.086	2.008
	2*σ	2.944	2.172	4.016
2 Seconds	Average	3.051	1.762	4.813
	Standard Deviation (σ)	2.820	2.151	3.876
	2*σ	5.640	4.302	7.751
3 Seconds	Average	4.696	2.743	7.440
	Standard Deviation (σ)	4.274	3.375	5.943
	2*σ	8.549	6.750	11.887
4 Seconds	Average	6.537	3.837	10.374
	Standard Deviation (σ)	5.911	4.762	8.311
	2*σ	11.821	9.525	16.621
5 Seconds	Average	8.553	5.027	13.580
	Standard Deviation (σ)	7.688	6.252	10.921
	2*σ	15.376	12.504	21.842

Table 5-8 Error analysis of test scenario 4

		Speed (Meters per Sec)
1 Second	Average	0.376
	Standard Deviation (σ)	0.397
	2*σ	0.794
2 Seconds	Average	0.584
	Standard Deviation (σ)	0.625
	2*σ	1.250
3 Seconds	Average	0.811
	Standard Deviation (σ)	0.892
	2*σ	1.783
4 Seconds	Average	1.067
	Standard Deviation (σ)	1.210
	2*σ	2.419
5 Seconds	Average	1.354
	Standard Deviation (σ)	1.585
	2*σ	3.169

Table 5-9 Error analysis of test scenario 5

		Heading (radians)
1 Second	Average	0.265
	Standard Deviation (σ)	0.915
	2*σ	1.831
2 Seconds	Average	0.419
	Standard Deviation (σ)	1.494
	2*σ	2.987
3 Seconds	Average	0.587
	Standard Deviation (σ)	2.199
	2*σ	4.398
4 Seconds	Average	0.777
	Standard Deviation (σ)	3.045
	2*σ	6.091
5 Seconds	Average	0.983
	Standard Deviation (σ)	4.026
	2*σ	8.051

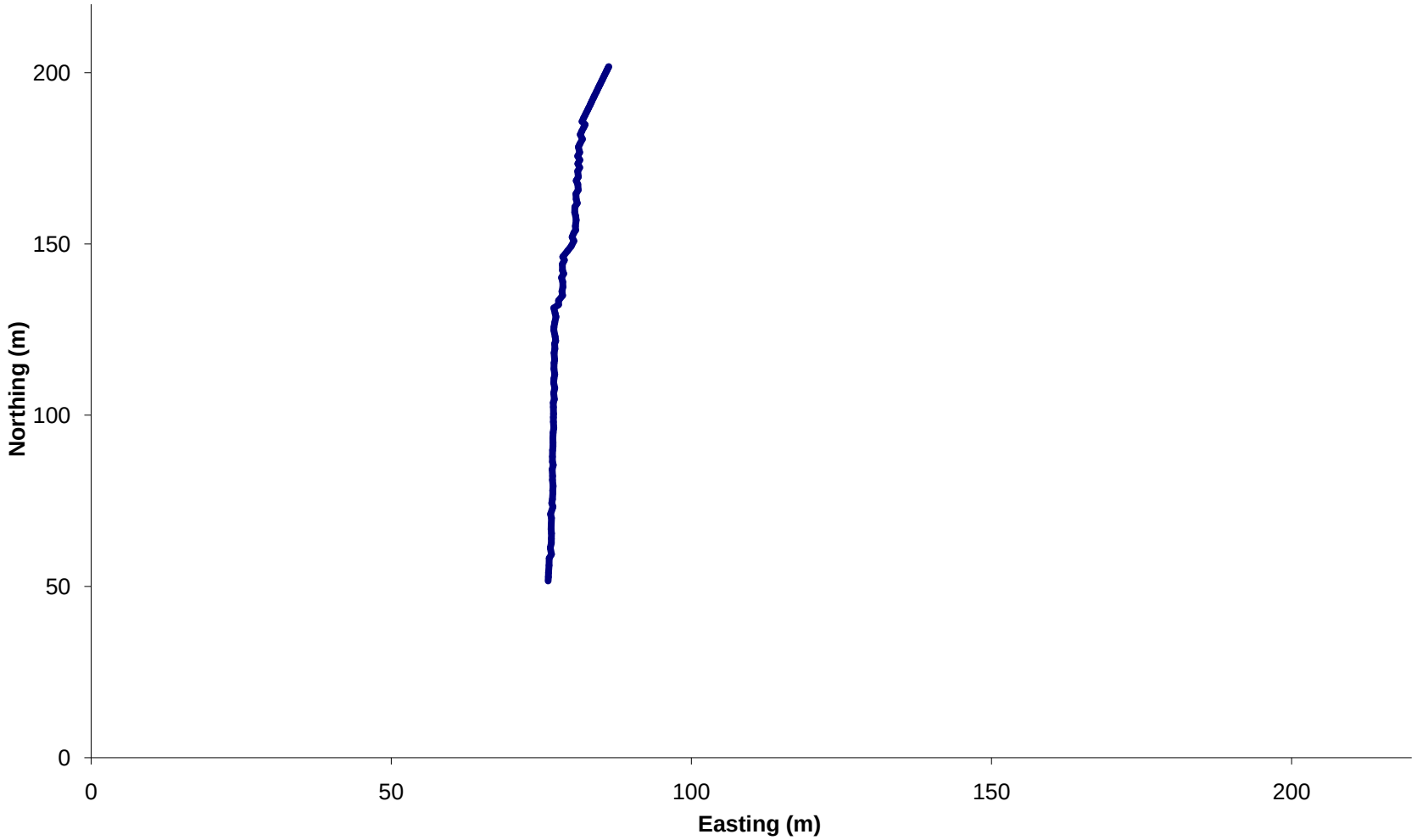

Figure 5-6 Easting vs. Northing Plot for Test Scenario 1.

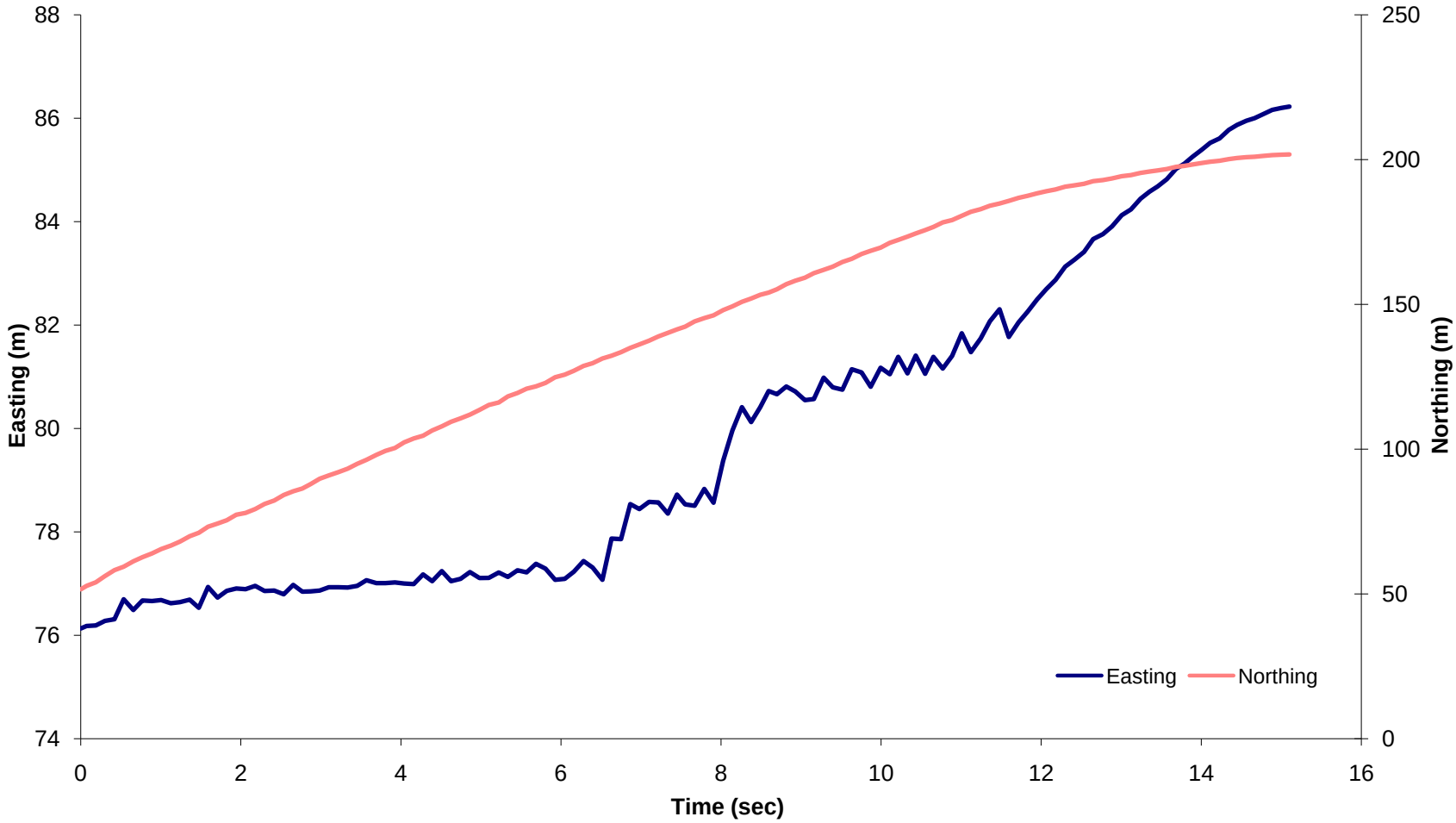

Figure 5-7 Easting and Northing vs. Time Plot for Test Scenario 1.

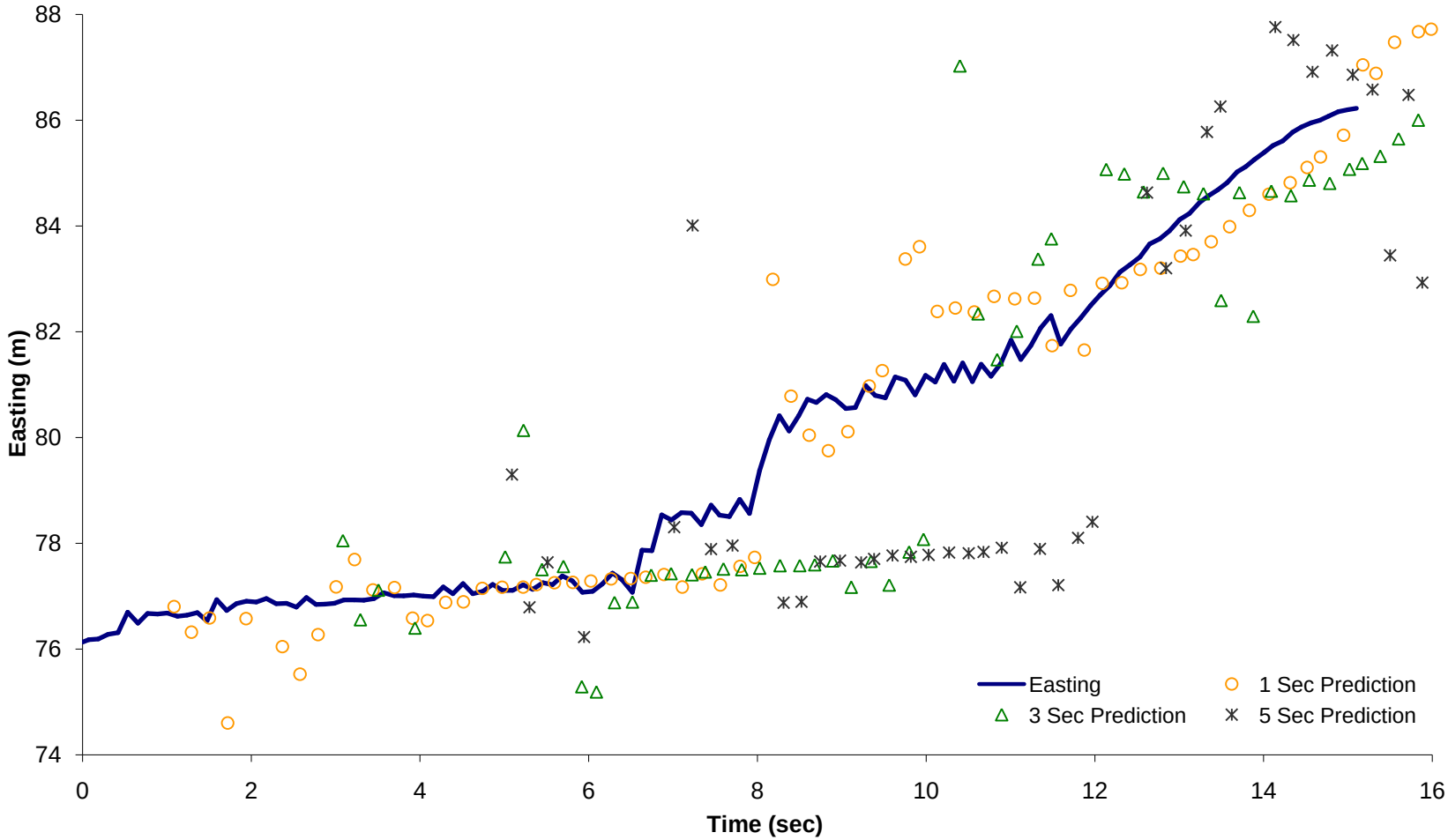

Figure 5-8 Easting vs. Time Plot with Prediction Values for Test Scenario 1.

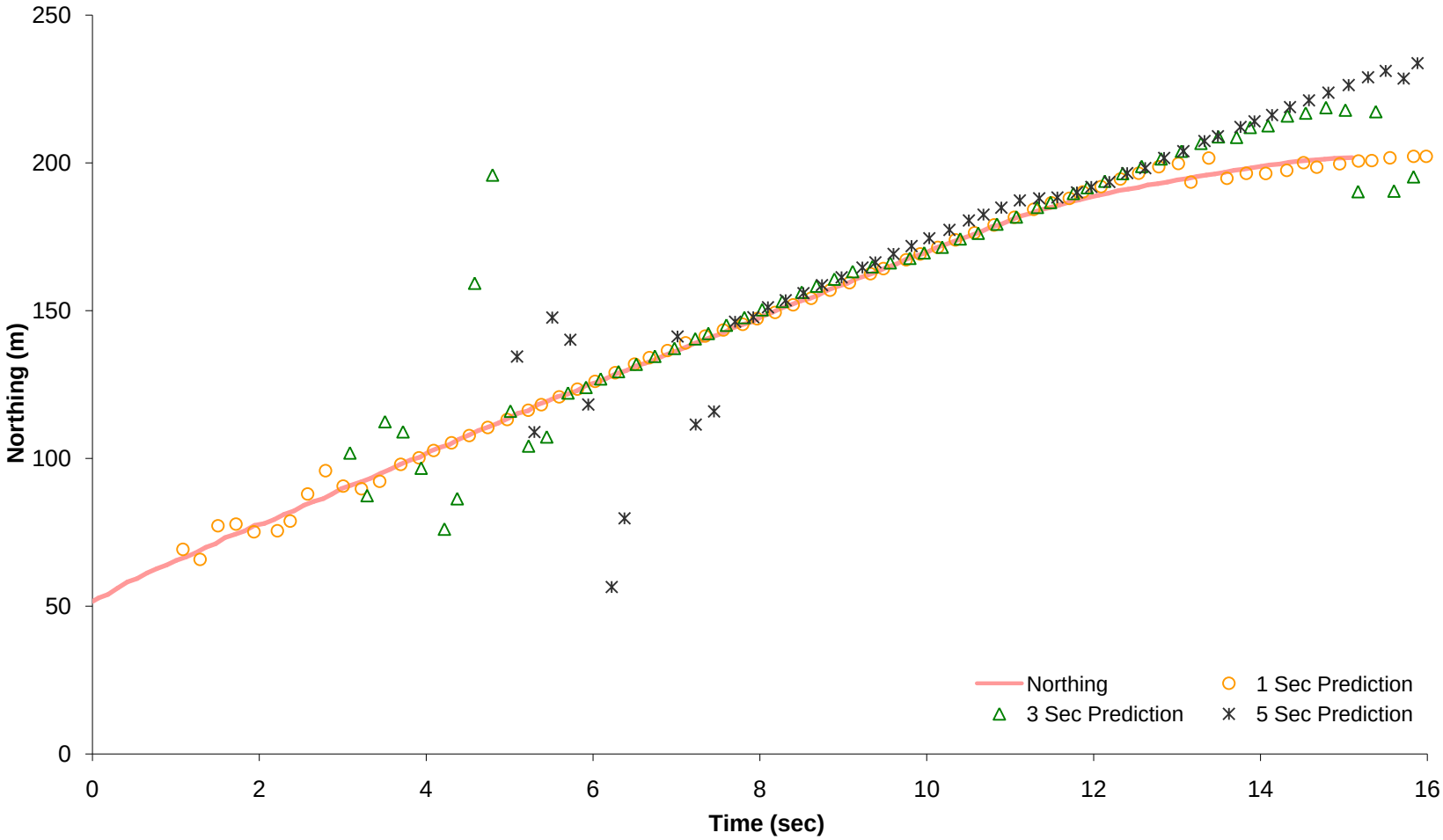

Figure 5-9 Northing vs. Time with Prediction Values for Test Scenario 1.

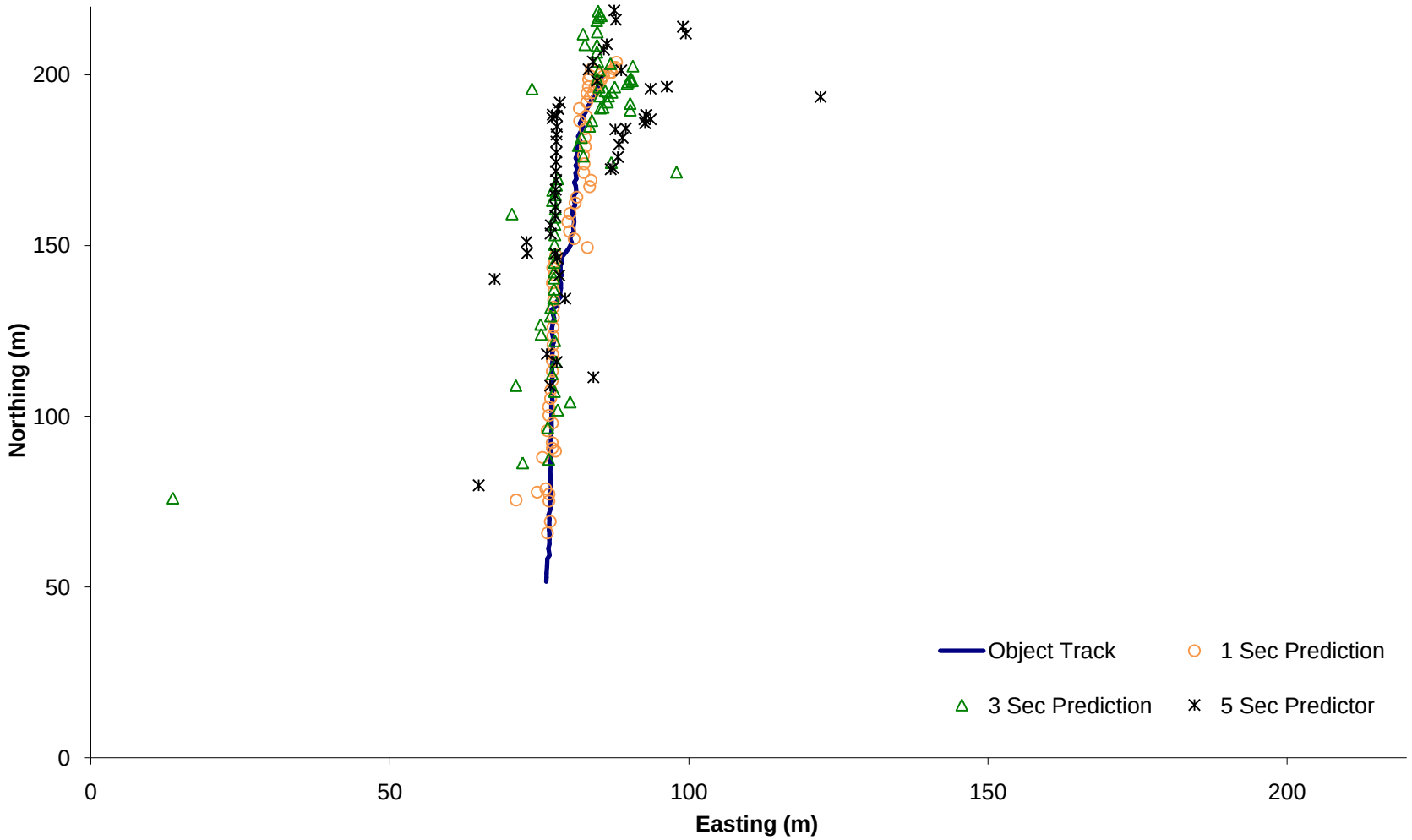

Figure 5-10 Easting vs. Northing with Prediction Values for Test Scenario 1.

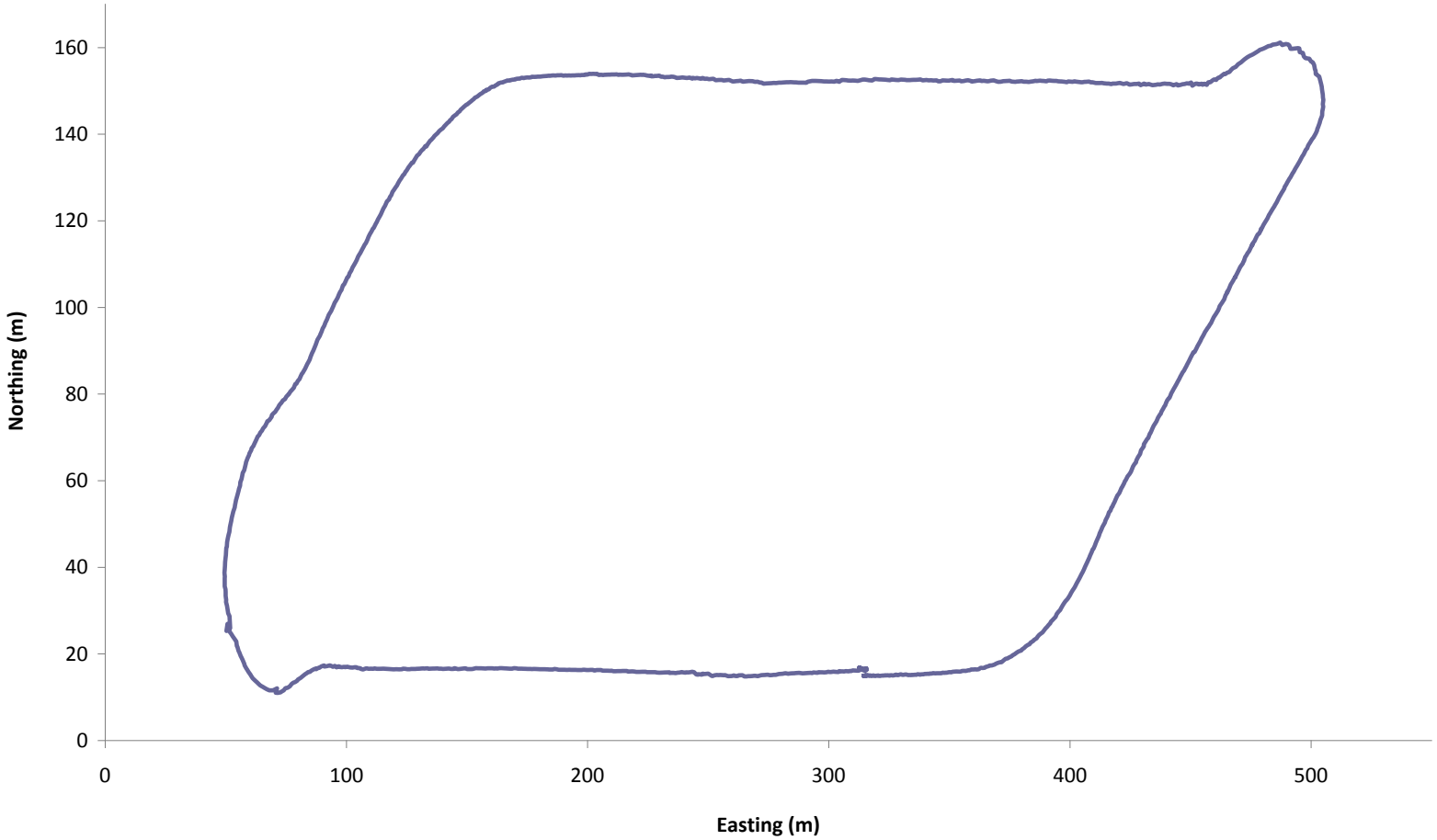

Figure 5-11 Easting vs. Northing for Test Scenario 2.

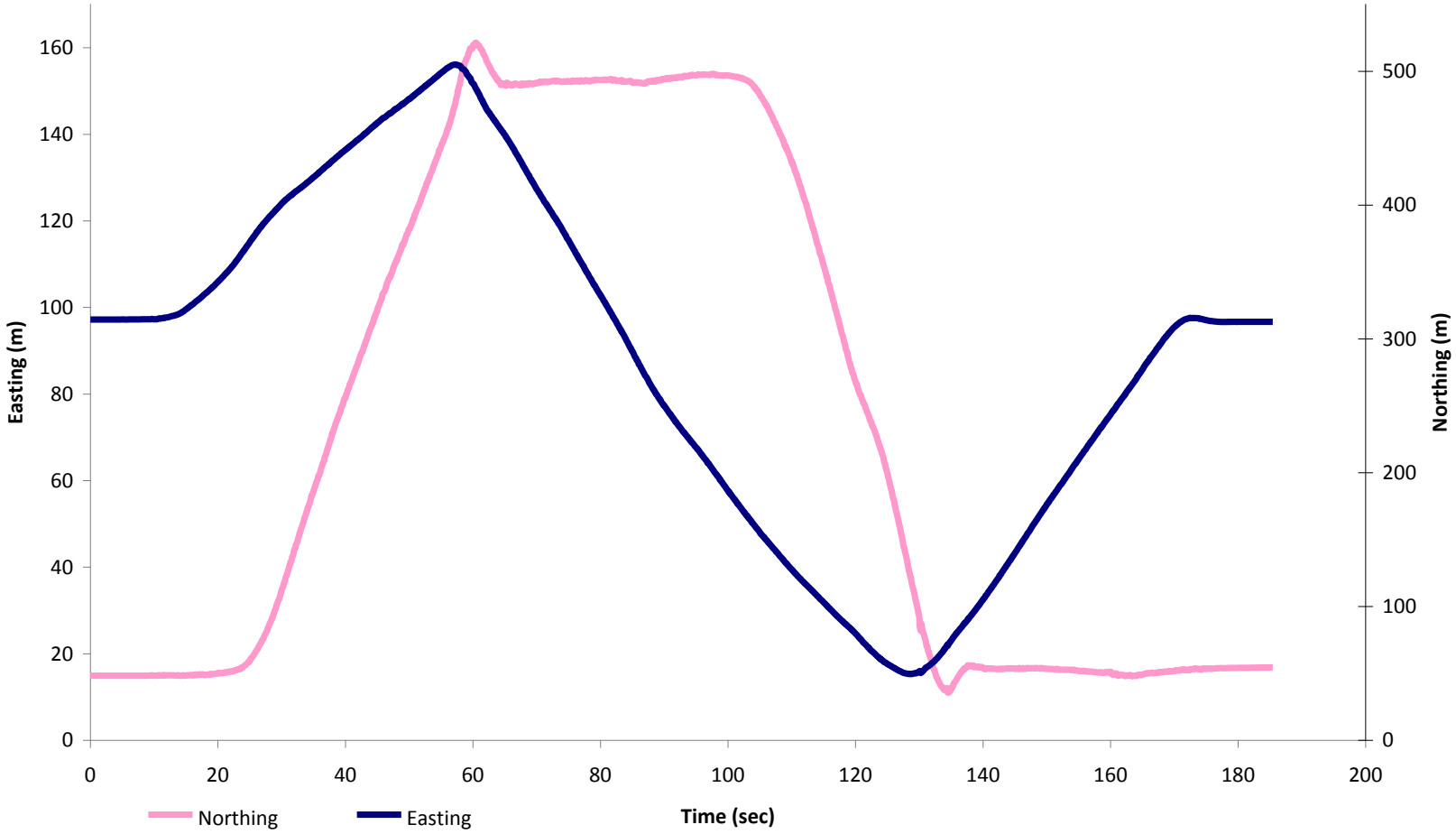

Figure 5-12 Easting and Northing vs. Time plot for Test Scenario 2.

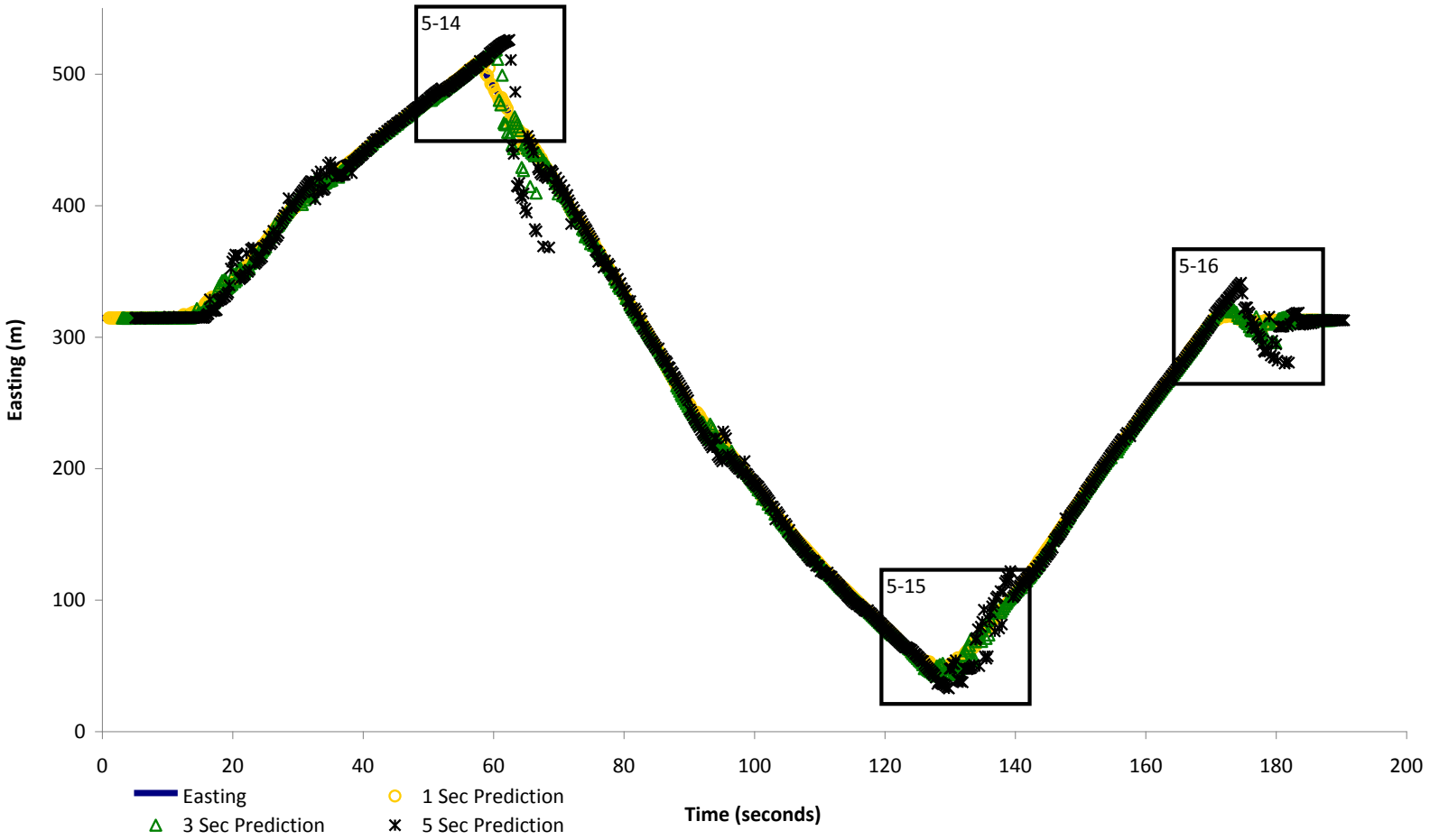

Figure 5-13 Easting vs. Time Plot for Test Scenario 2. Boxes correspond to close-ups of data which are shown in figures 5-14, 5-15 and 5-16 respectively.

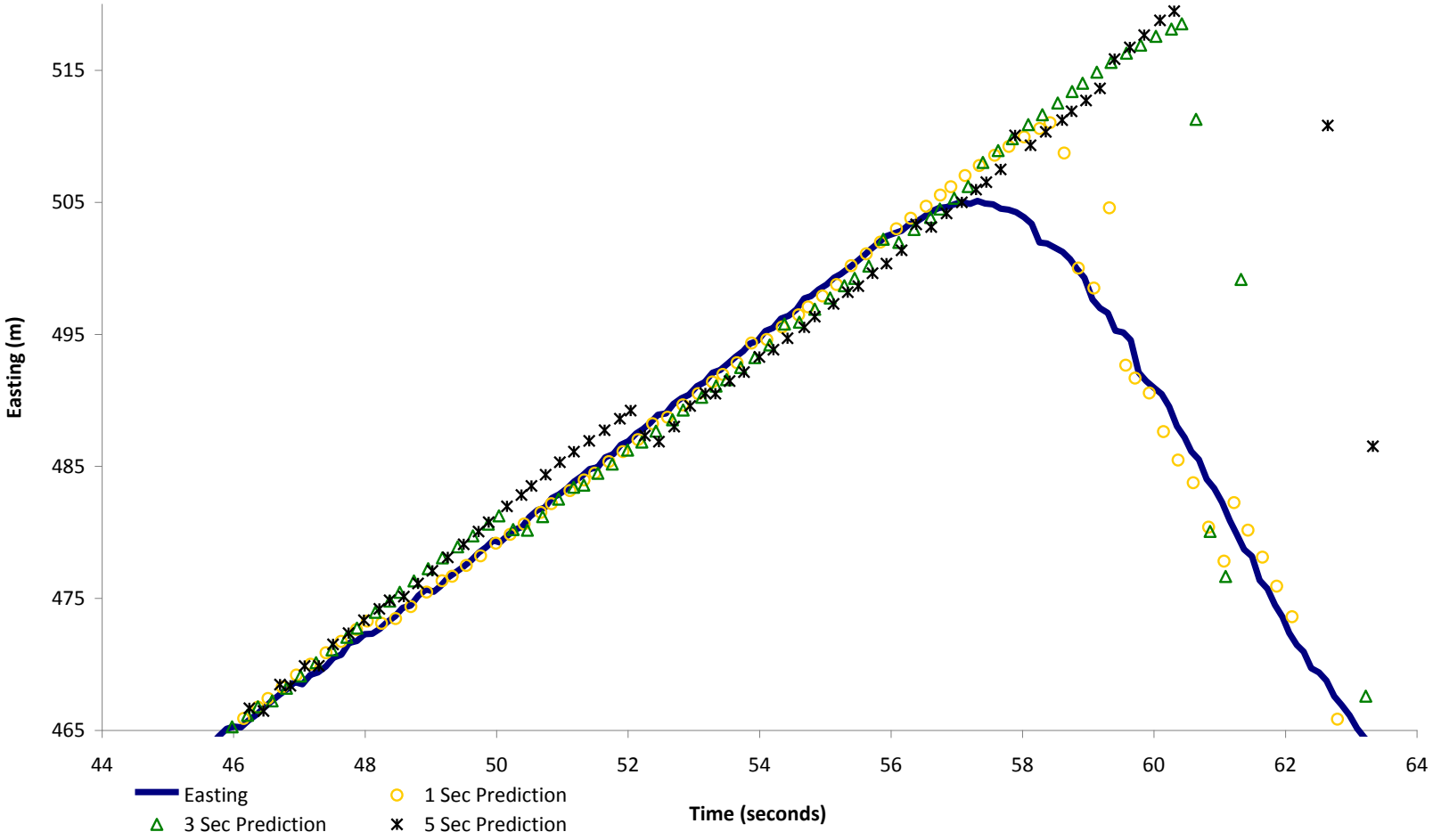

Figure 5-14 Easting vs. Time with prediction values for test scenario 2. Close up of time period 44-64 seconds.

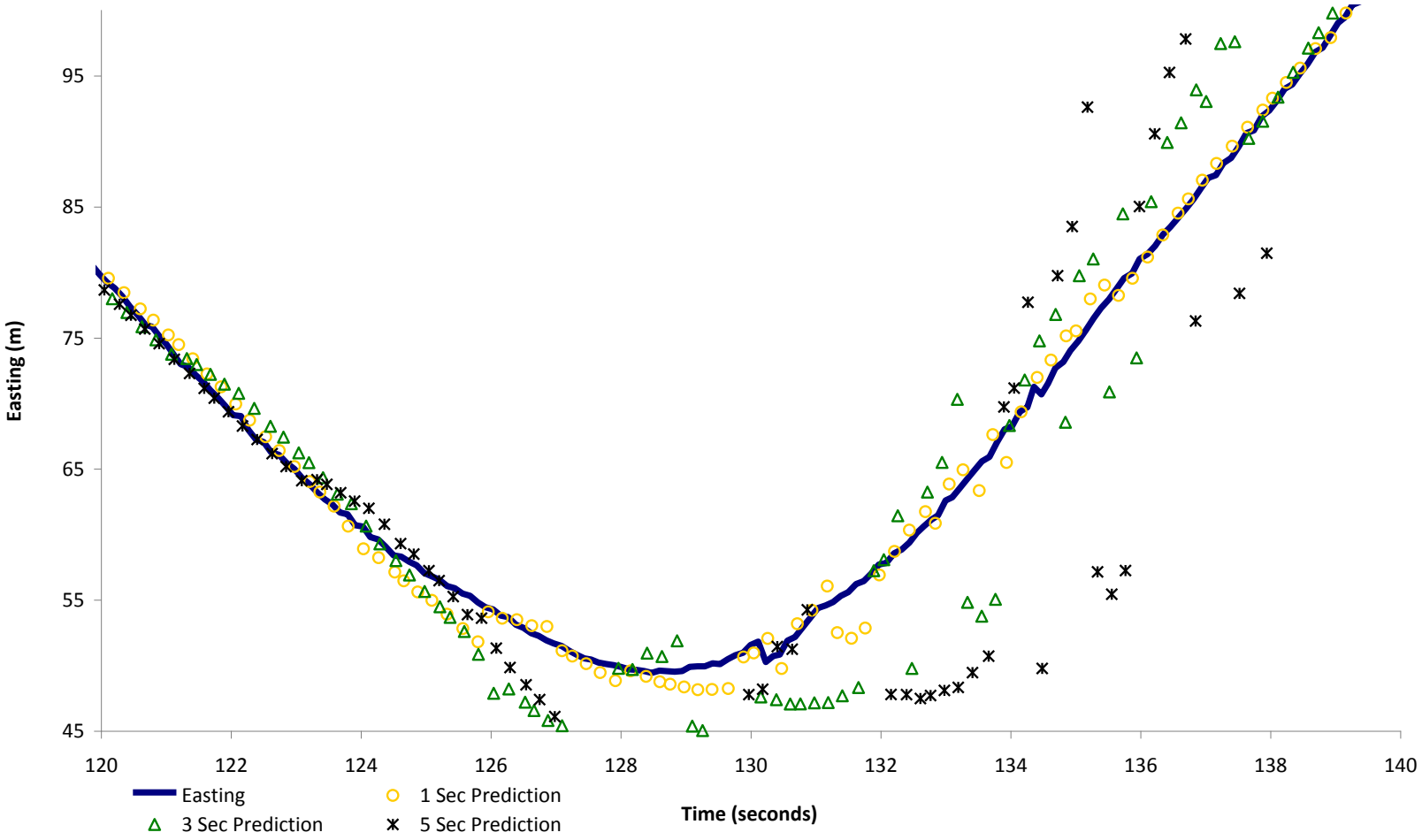

Figure 5-15 Easting vs. Time with prediction values for test scenario 2. Close up of time period 120-140 seconds.

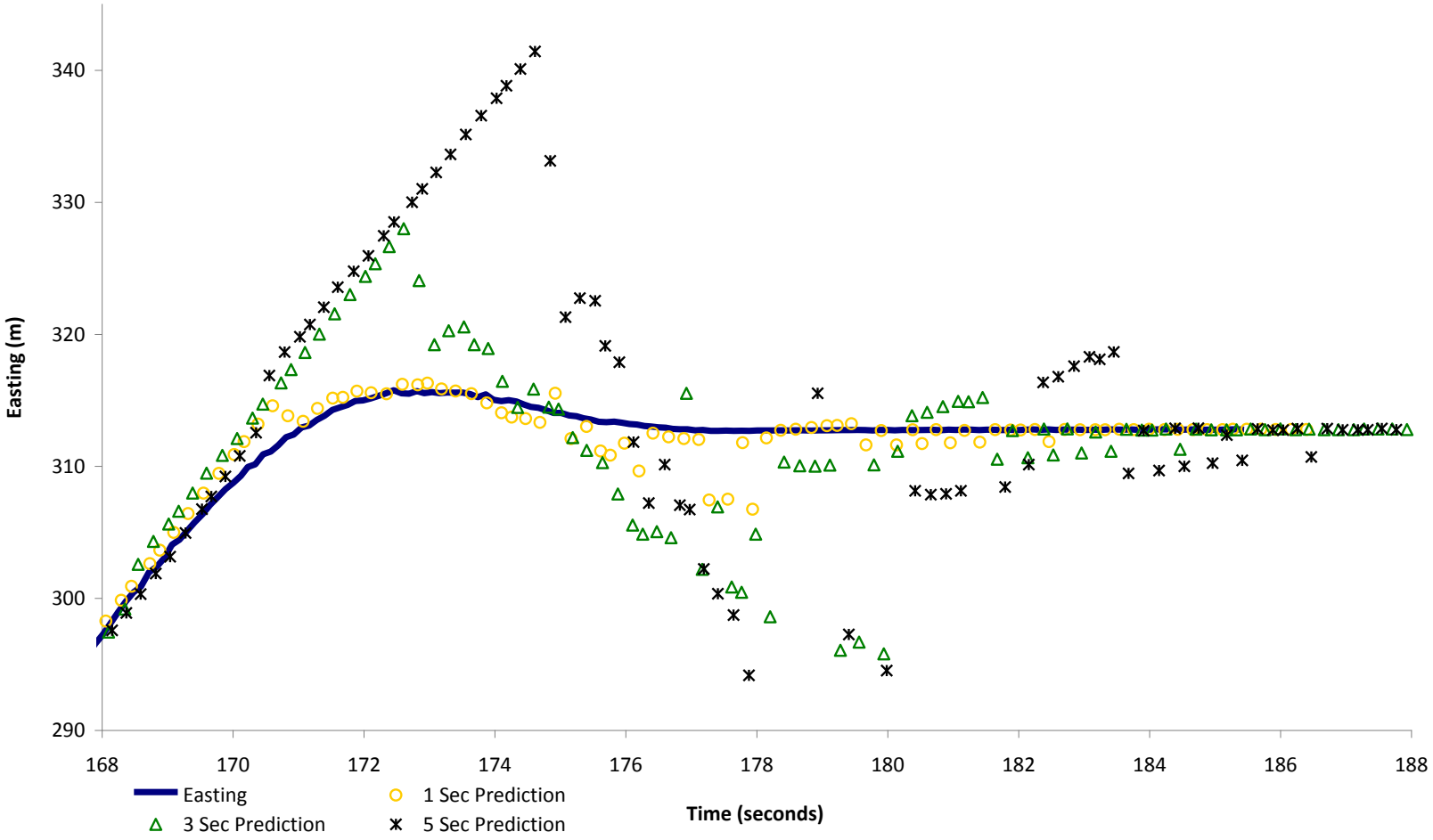

Figure 5-16 Easting vs. Time with prediction values for test scenario 2. Close up of time period 168-188 seconds.

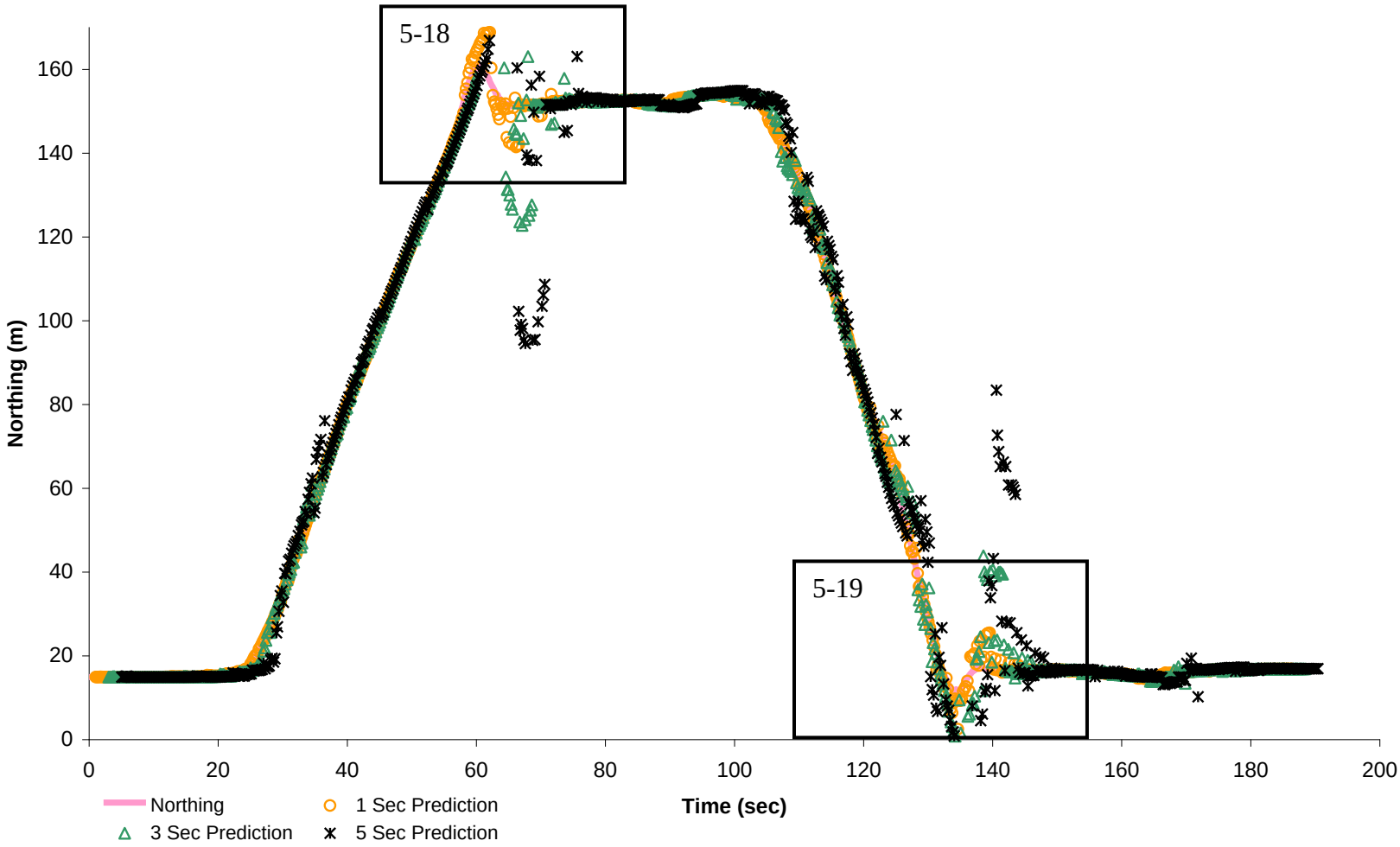

Figure 5-17 Northing vs. Time with prediction values for Test Scenario 2.

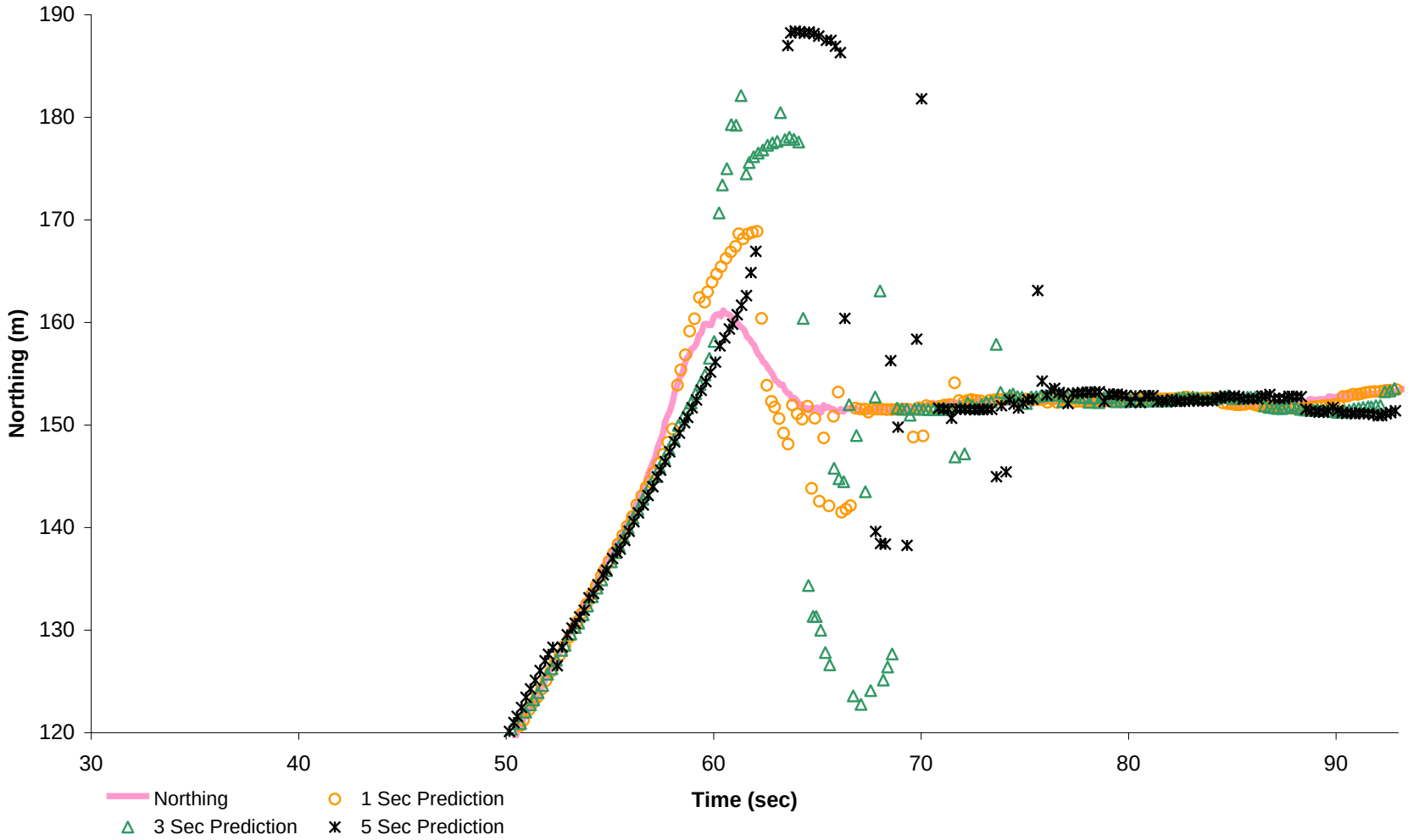

Figure 5-18 Northing vs. Time with prediction values for Test Scenario 2. Close up of time period 30-90 seconds.

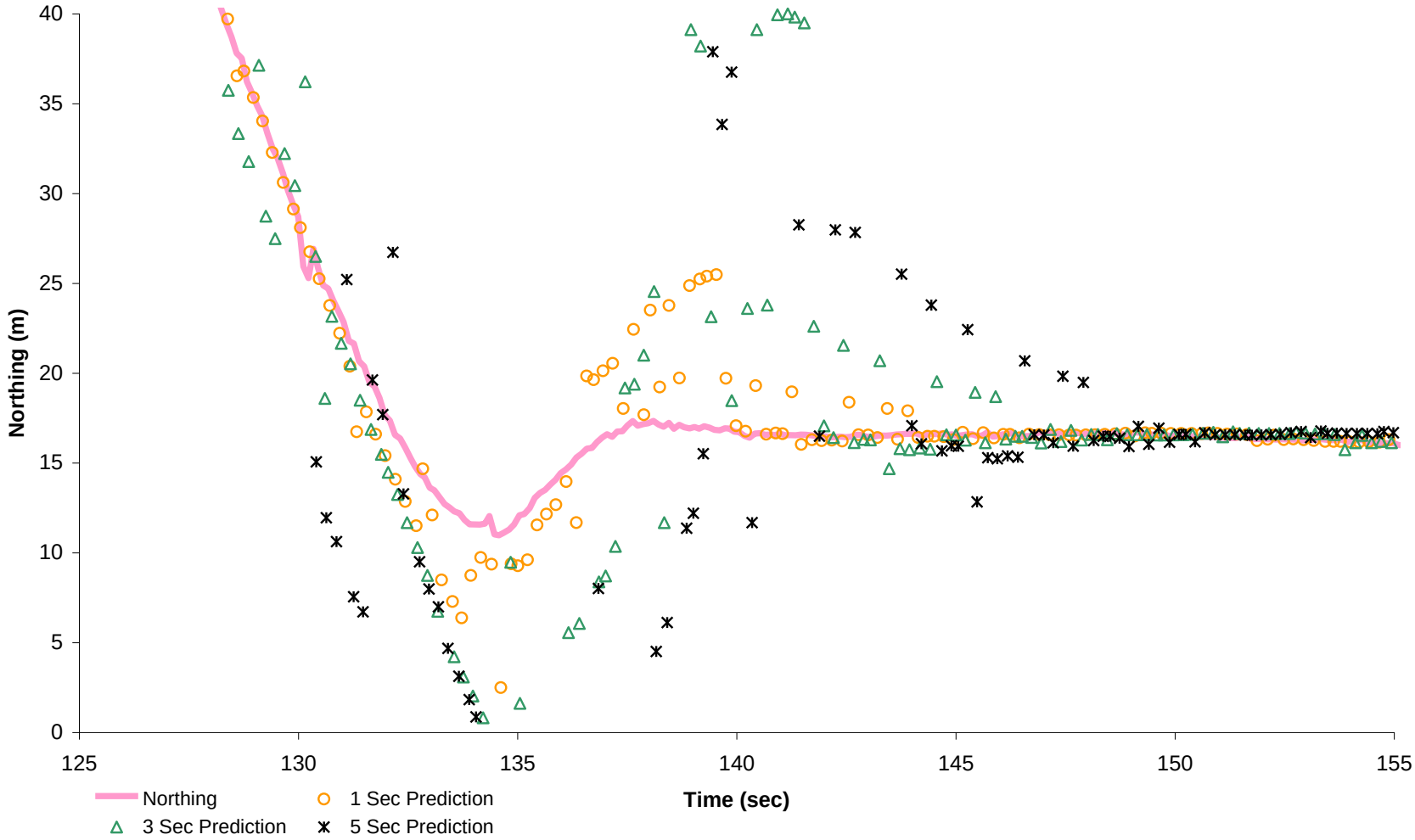

Figure 5-19 Northing vs. Time with prediction values for Test Scenario 2. Close up of time period 125-155 seconds.

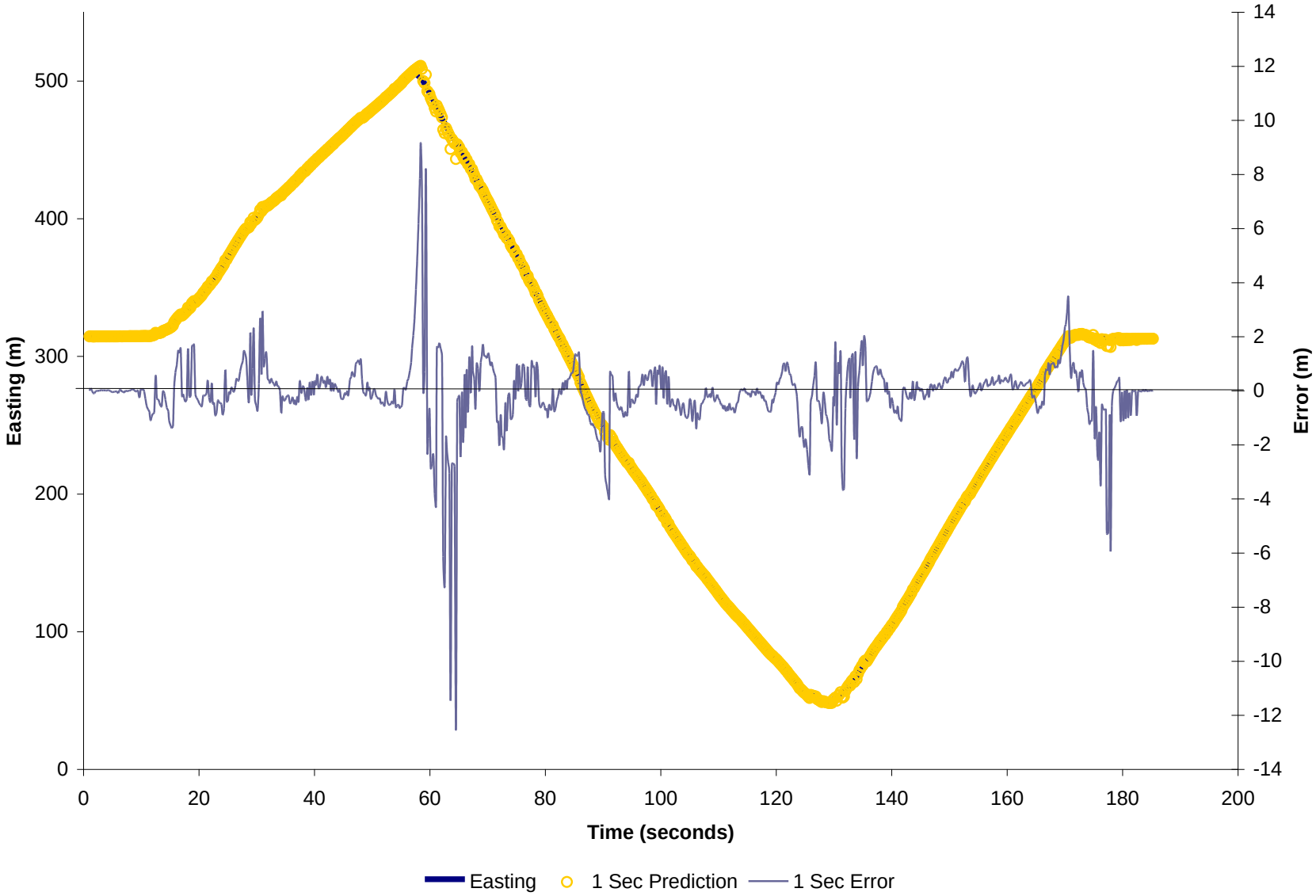

Figure 5-20 Easting Error vs. Time for test scenario 2.

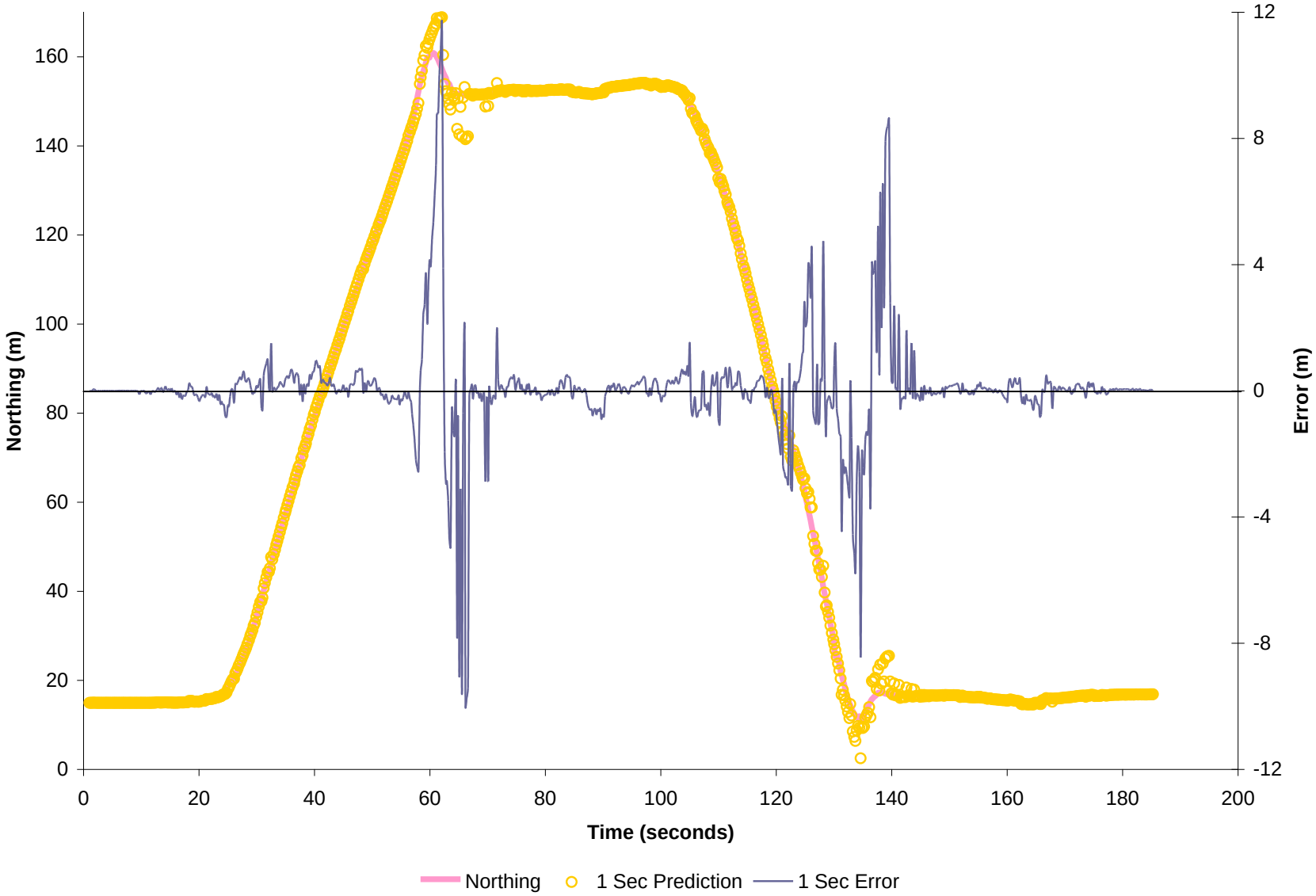

Figure 5-21 Northing Error vs. Time for test scenario 2.

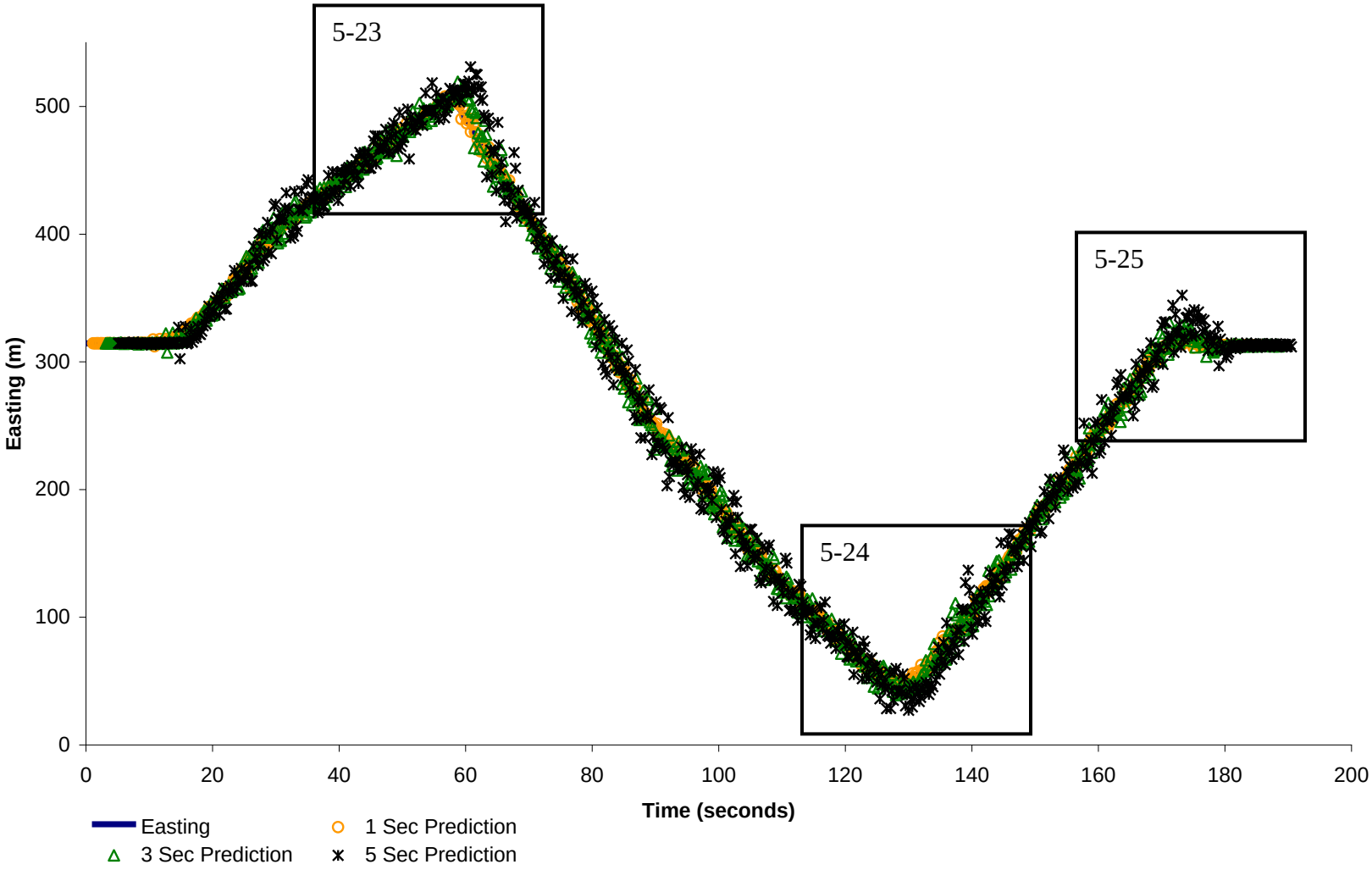

Figure 5-22 Easting vs. Time with prediction values for test scenario 3. Close up plots of critical sections are provided in figures 5-23, 5-24 and 5-25 as shown.

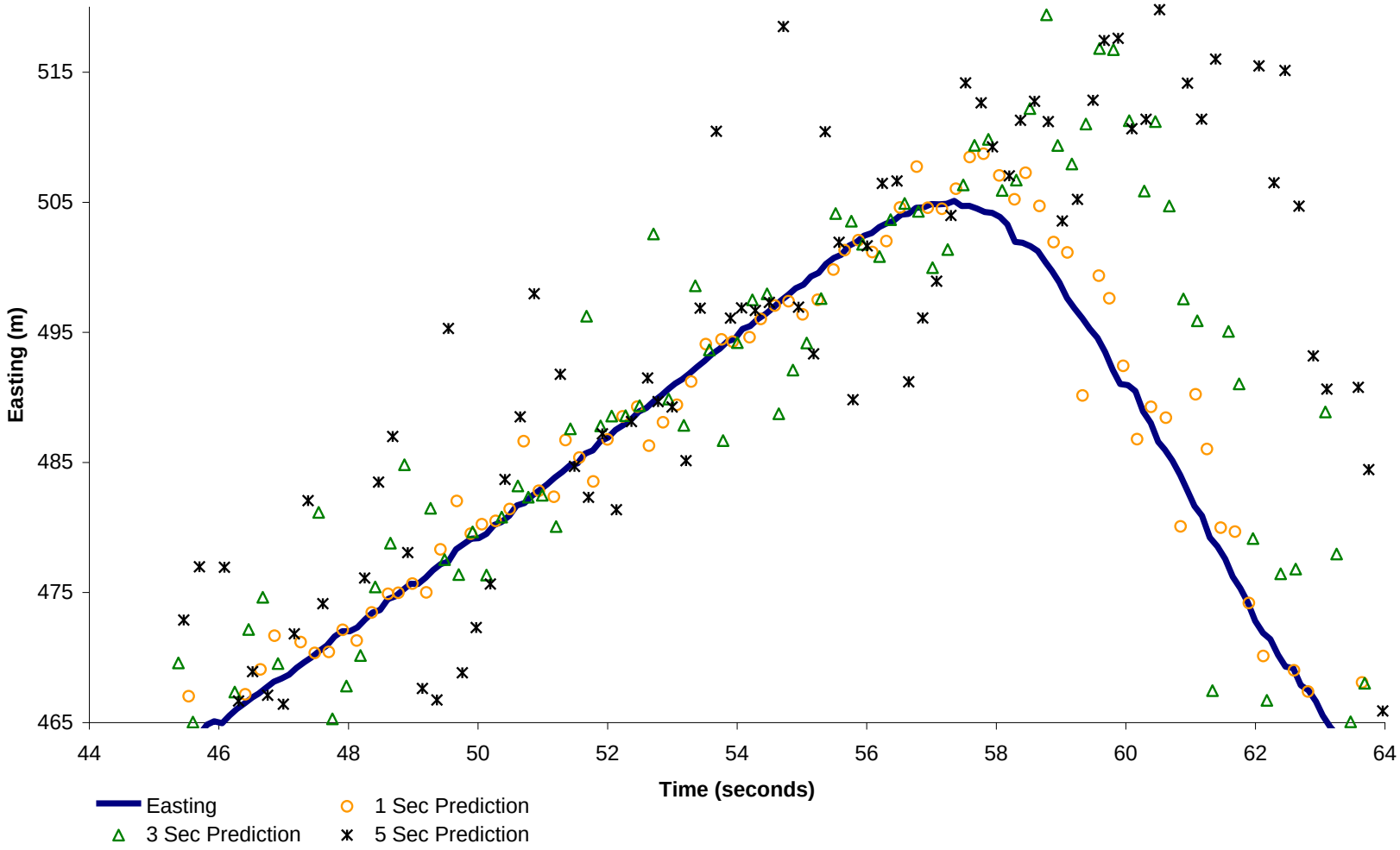

Figure 5-23 Easting vs. Time with prediction values for test scenario 3. Close up of time period 44-64 seconds.

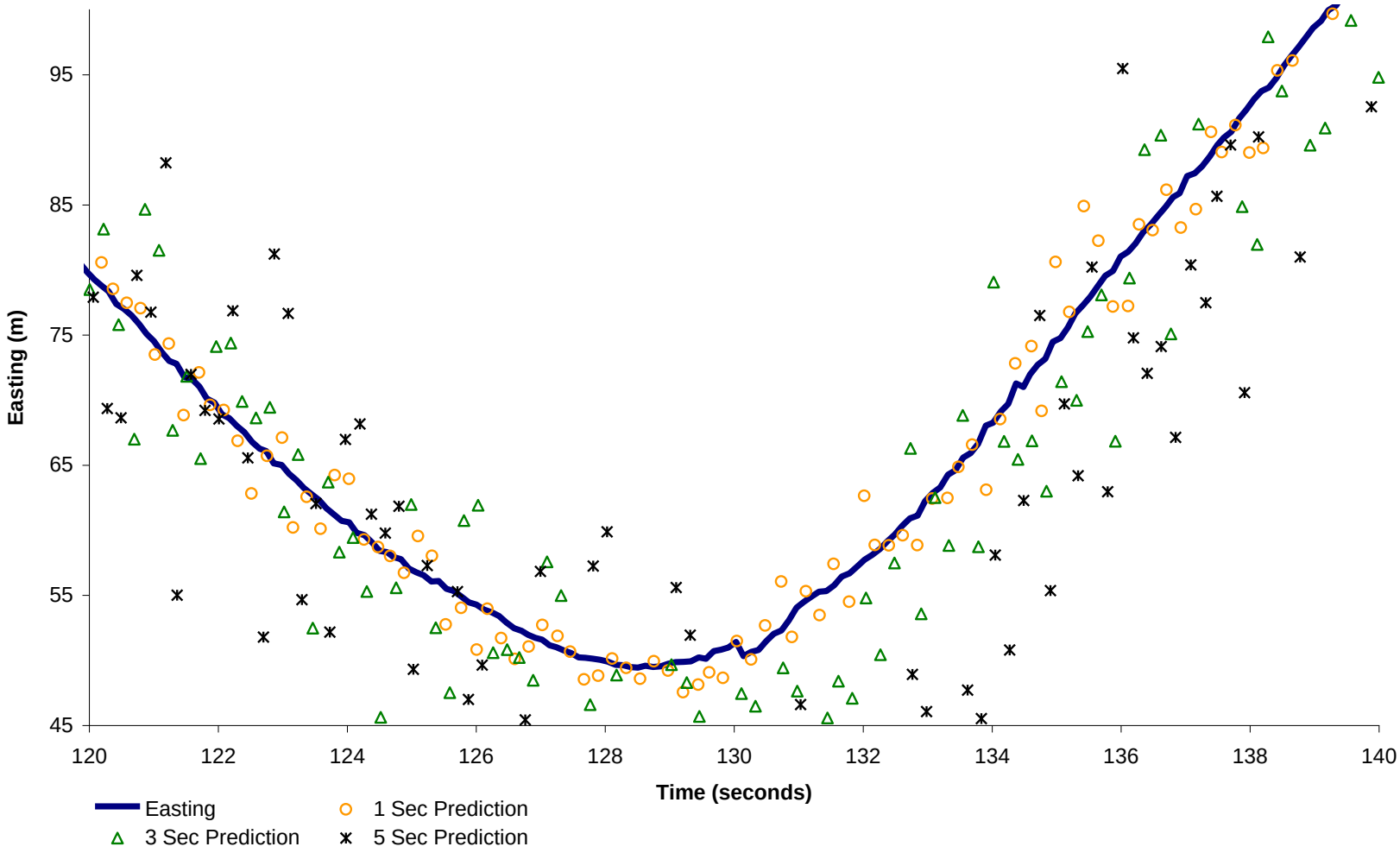

Figure 5-24 Easting vs. Time with prediction values for test scenario 3. Close up of time period 120-140 seconds.

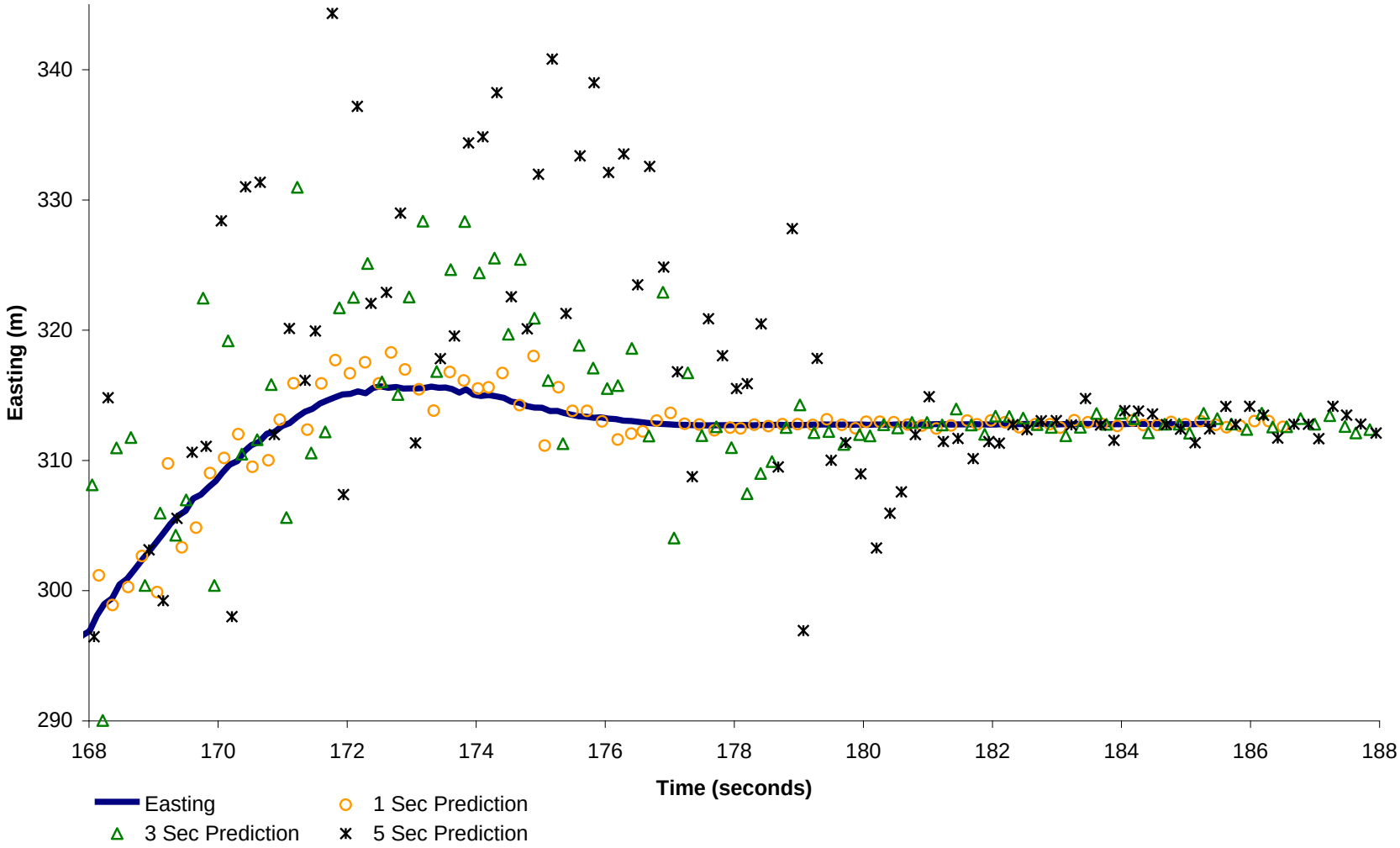

Figure 5-25 Easting vs. Time with prediction values for test scenario 3. Close up of time period 168-188 seconds.

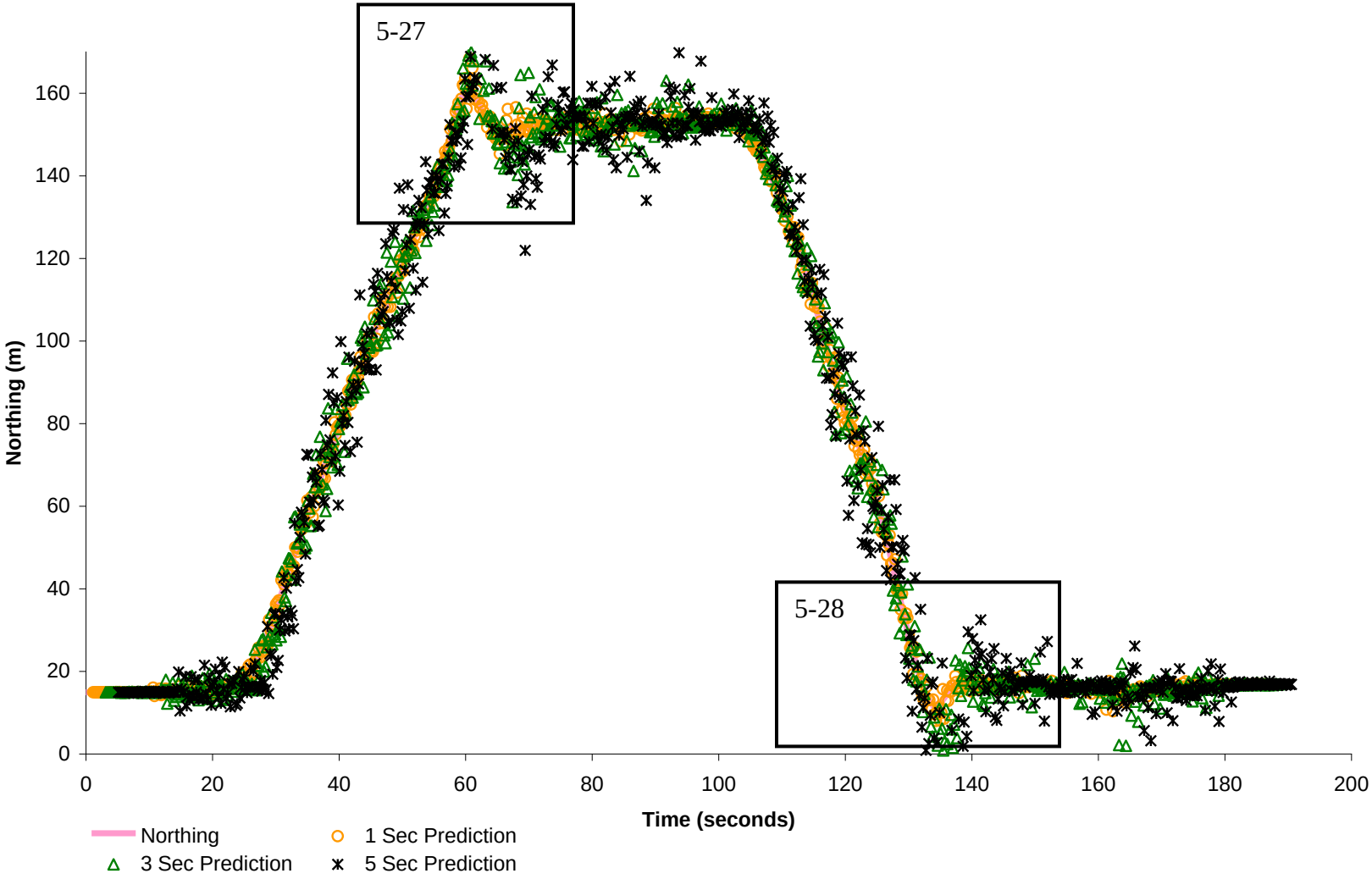

Figure 5-26 Northing vs. Time with predicted values for test scenario 3. Close ups of highlighted areas are provided in figures 5-27 and 5-28 as indicated.

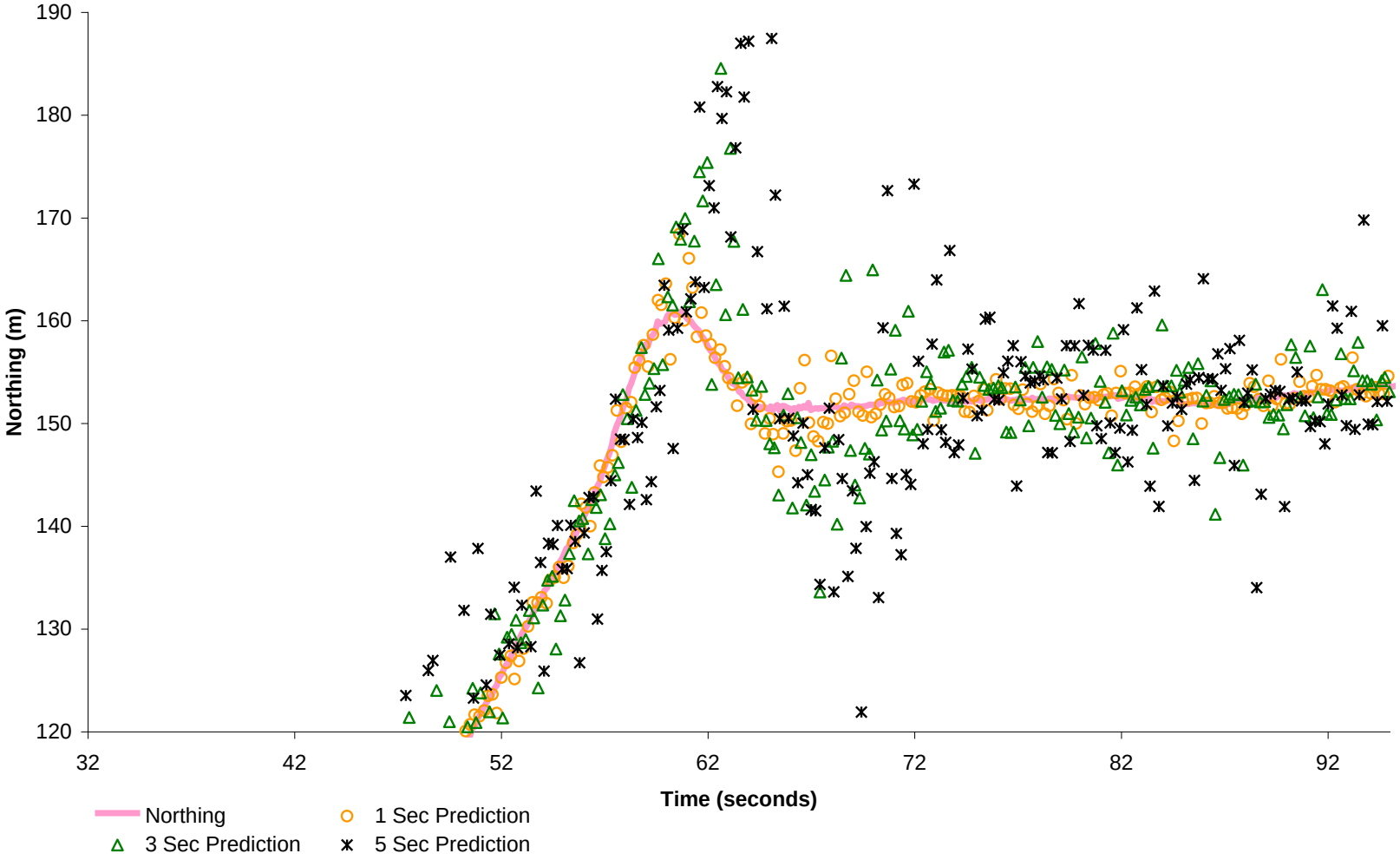

Figure 5-27 Northing vs. Time with predicted values for test scenario 3. Close up of time period 32-95 seconds.

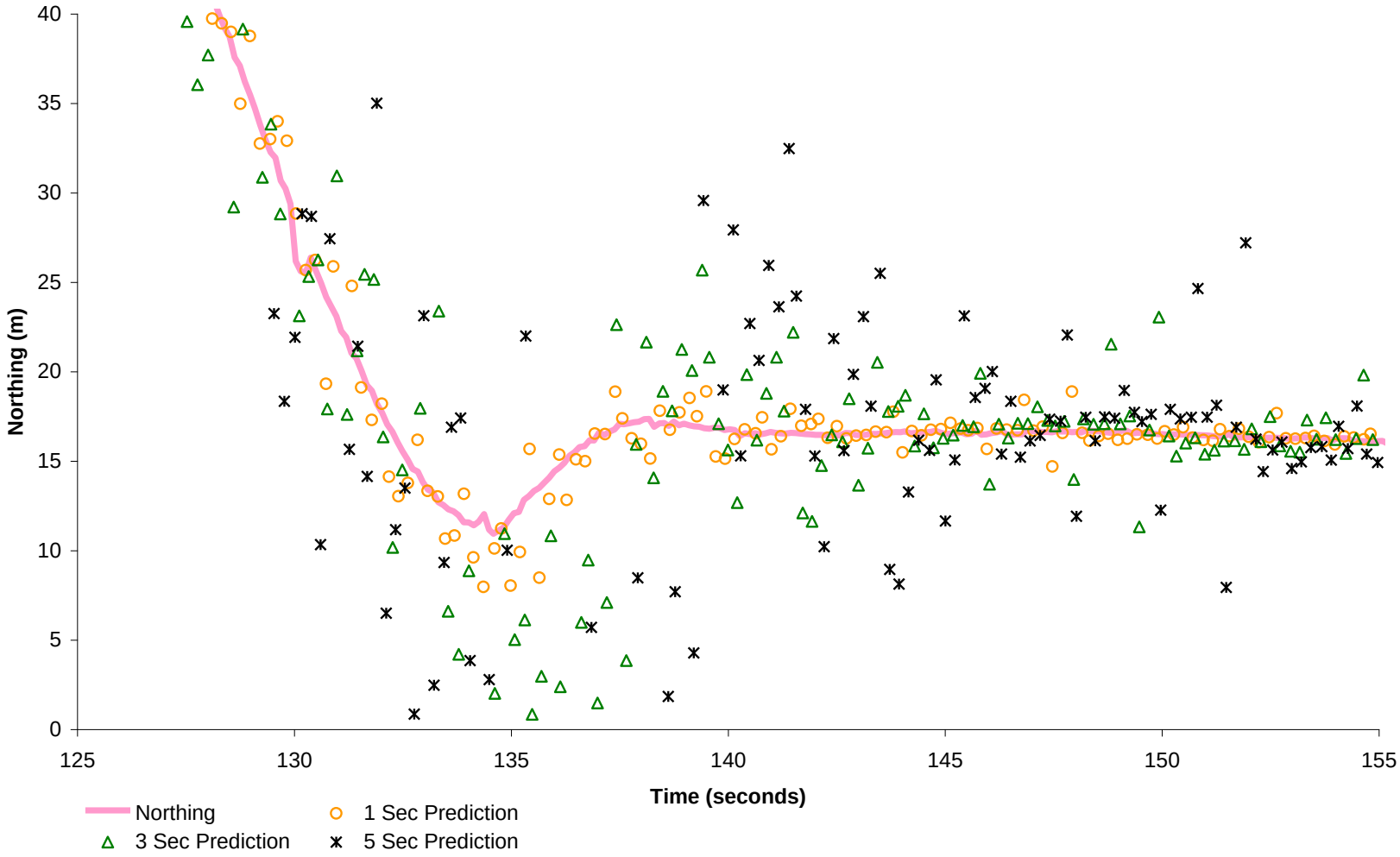

Figure 5-28 Northing vs. Time with predicted values for test scenario 3. Close up of time period 125-155 seconds.

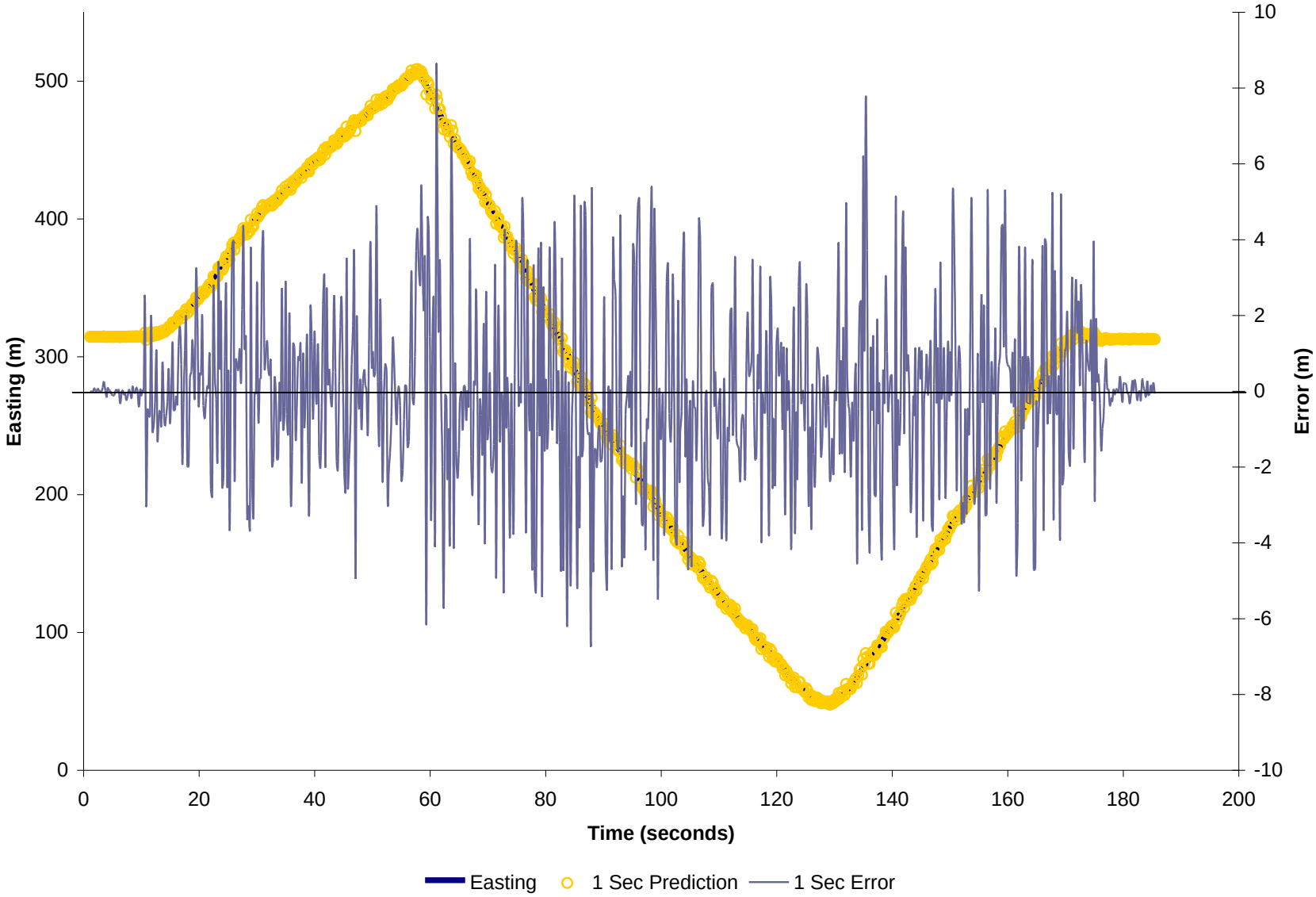

Figure 5-29 Easting Error vs. Time for test scenario 3.

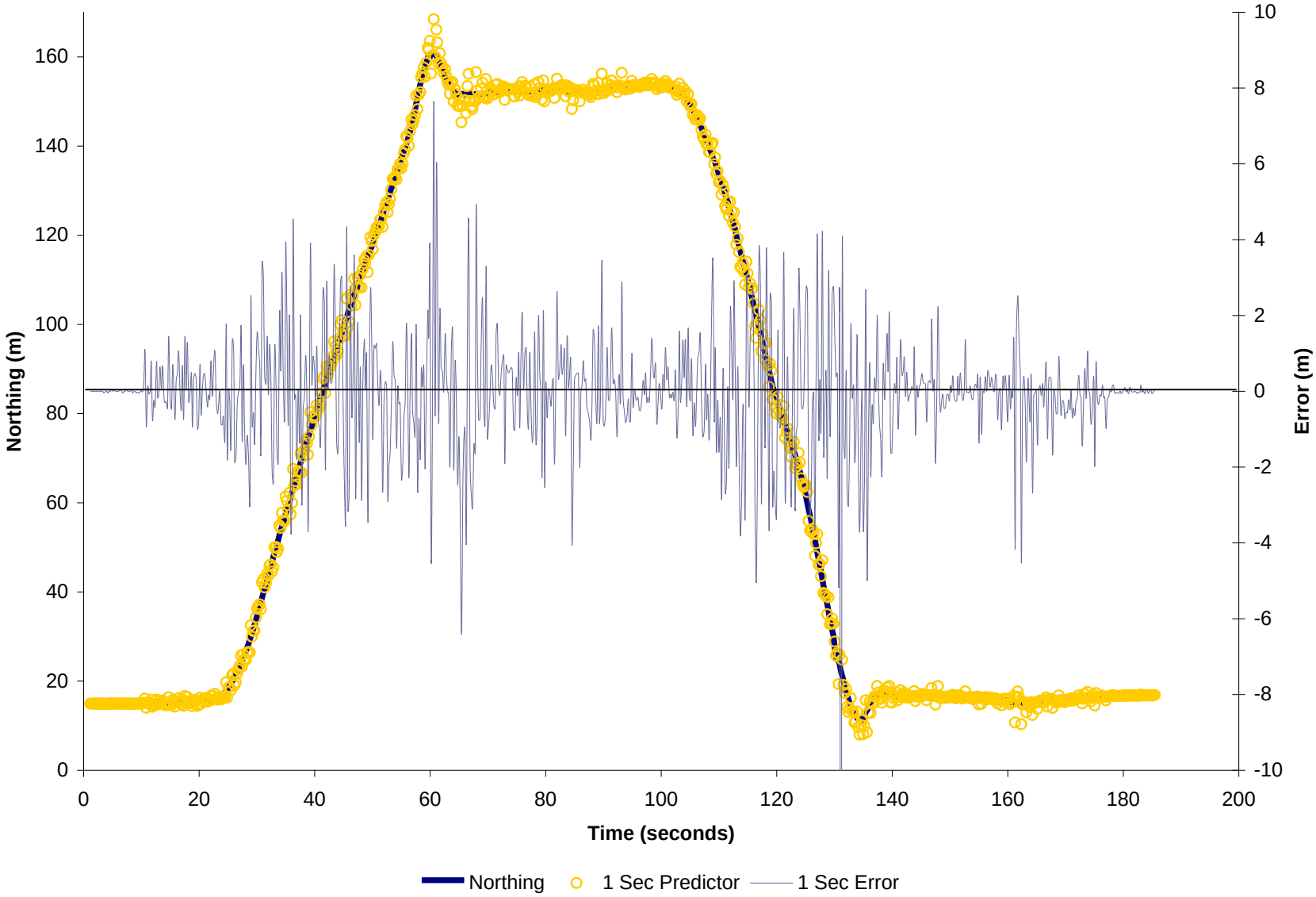

Figure 5-30 Northing Error vs. Time for test scenario 3.

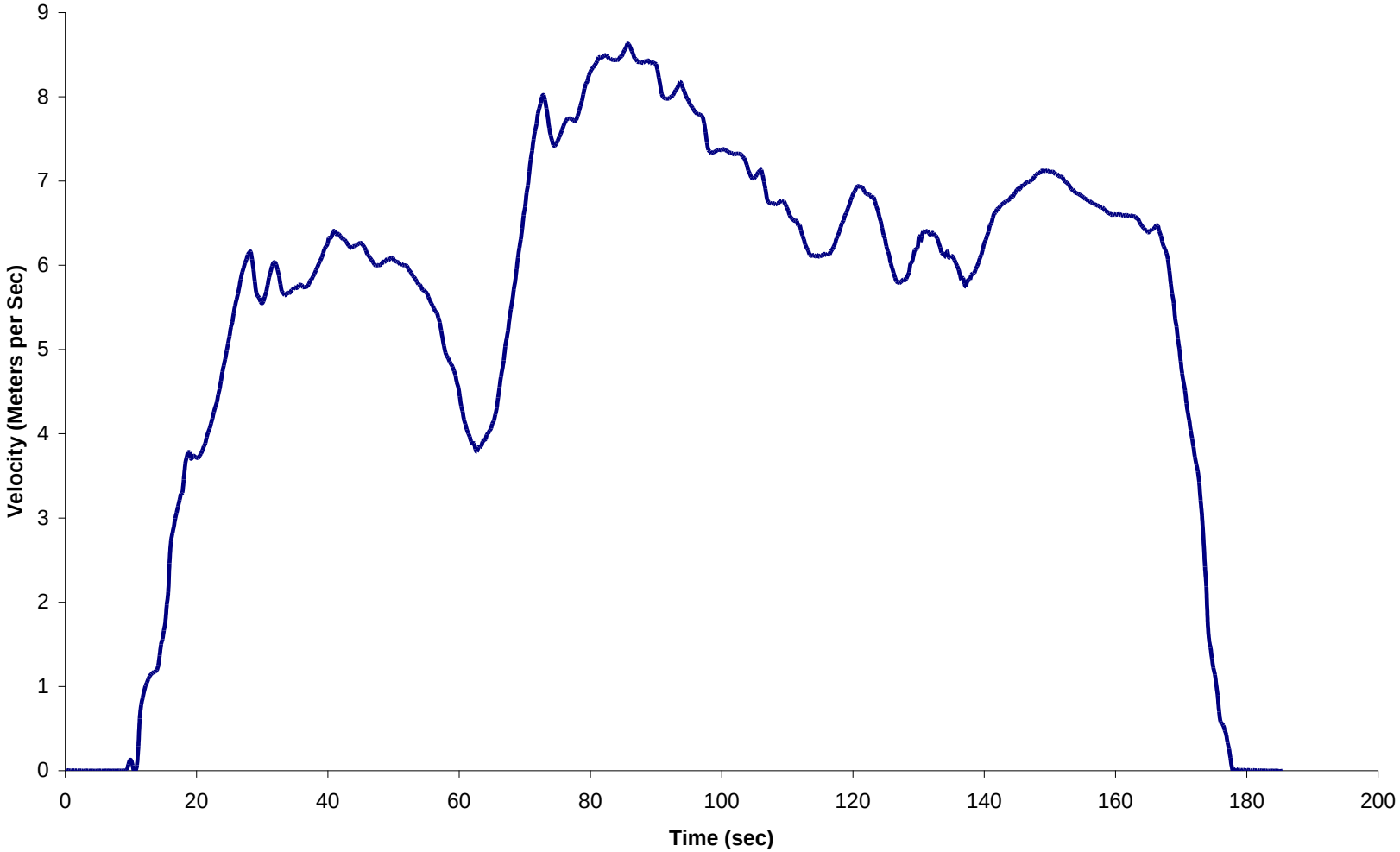

Figure 5-31 Velocity vs. Time plot for test scenario 4.

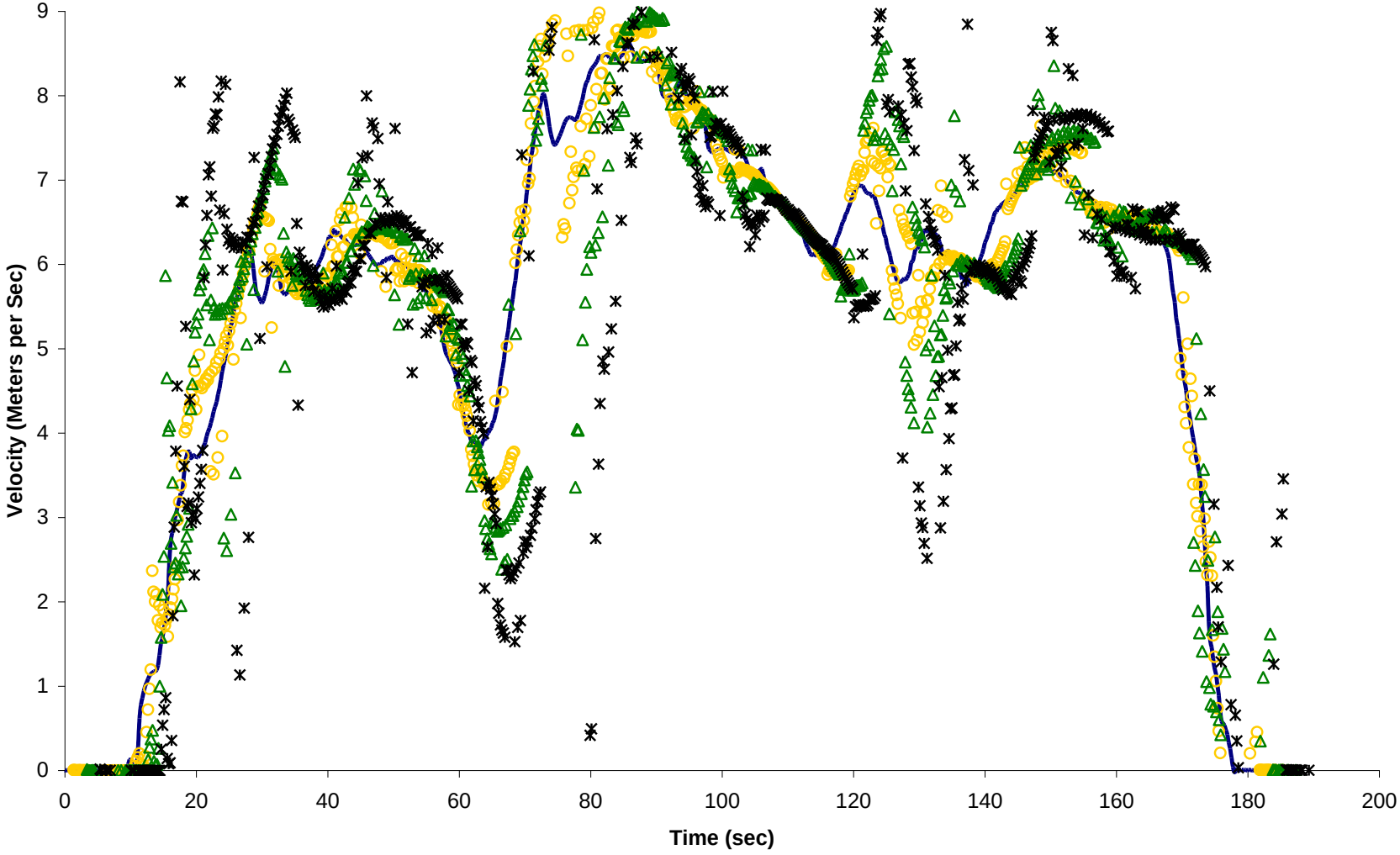

Figure 5-32 Velocity vs. Time plot with prediction values for test scenario 4.

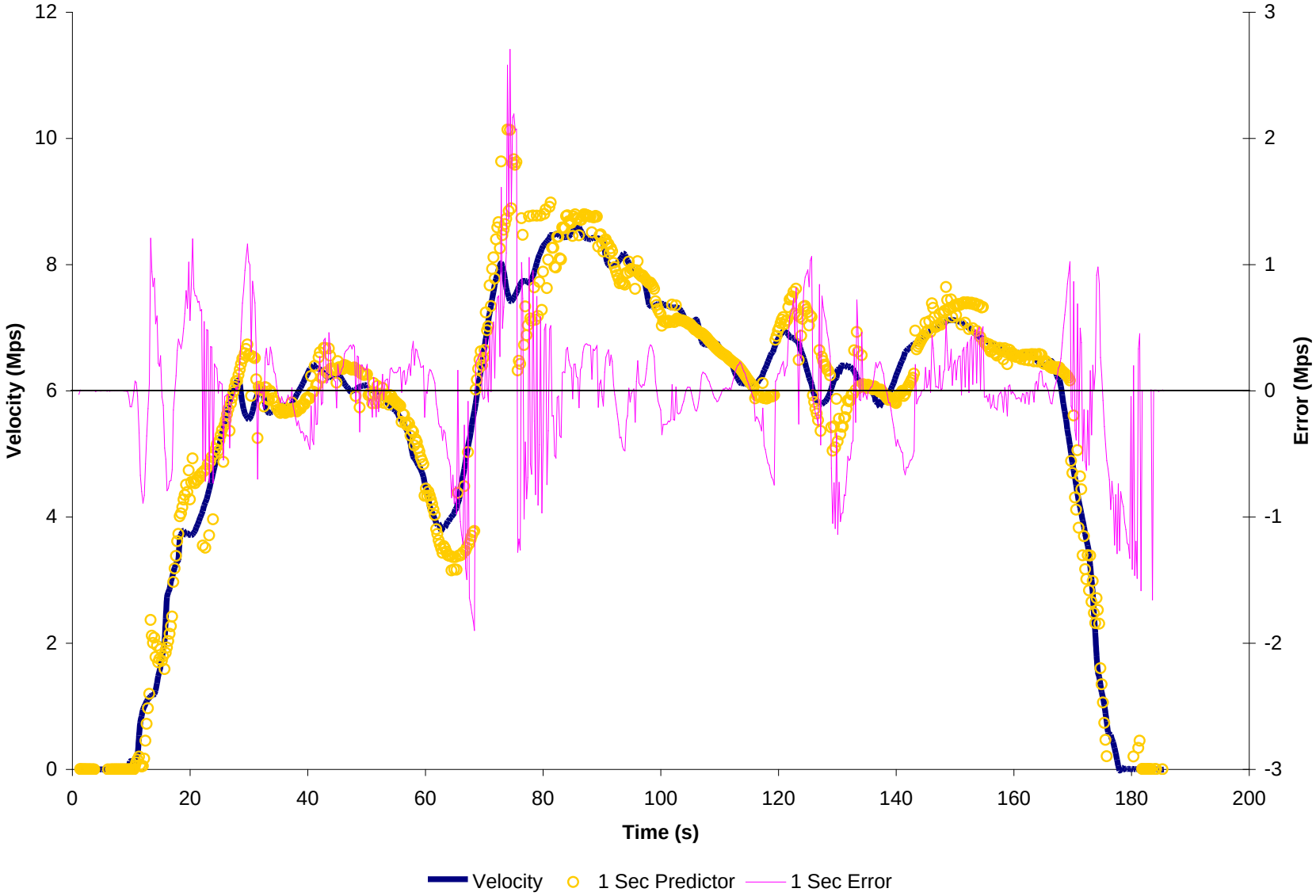

Figure 5-33 Error vs Time for test scenario 4.

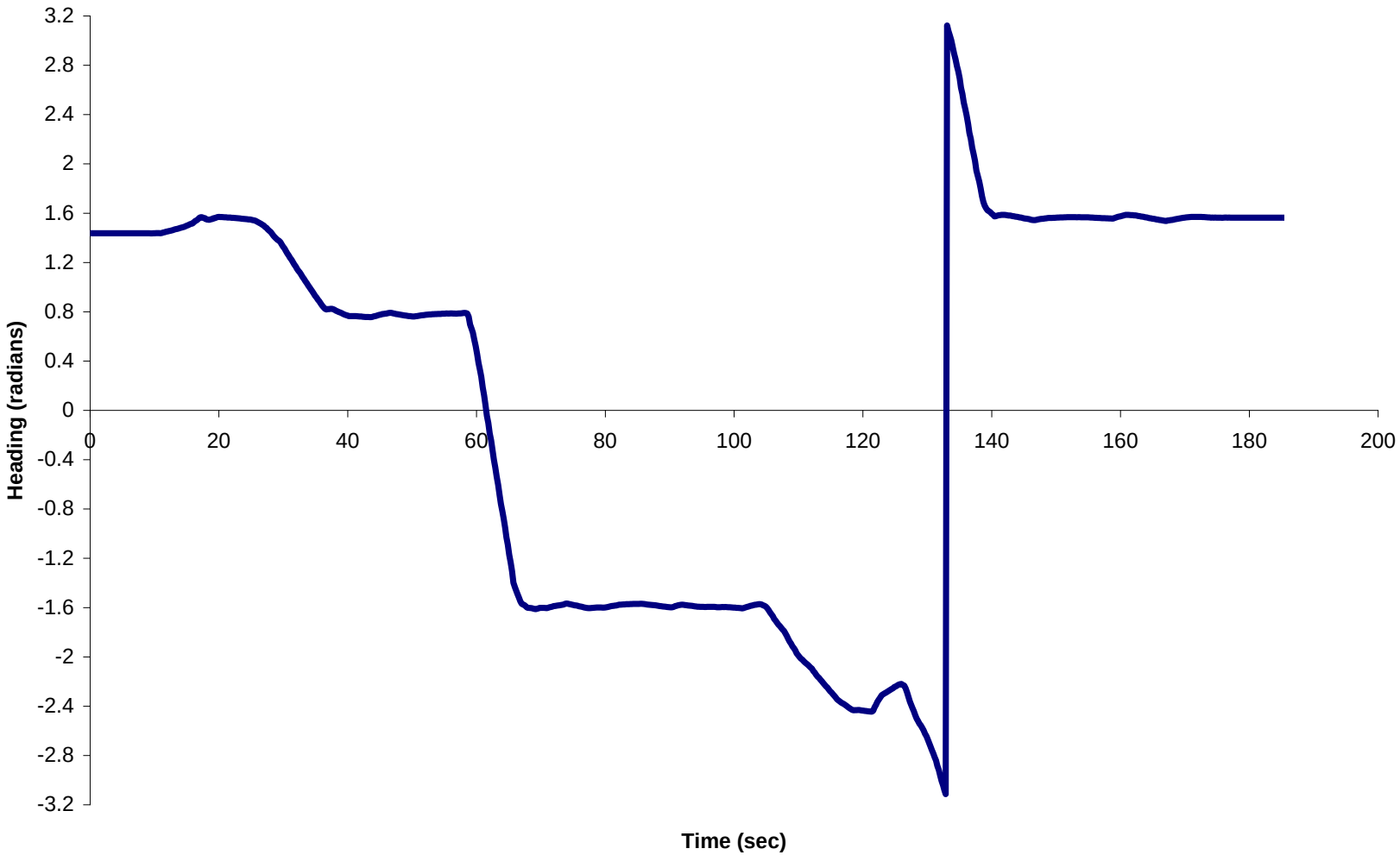

Figure 5-34 Heading vs. Time plot for test scenario 5.

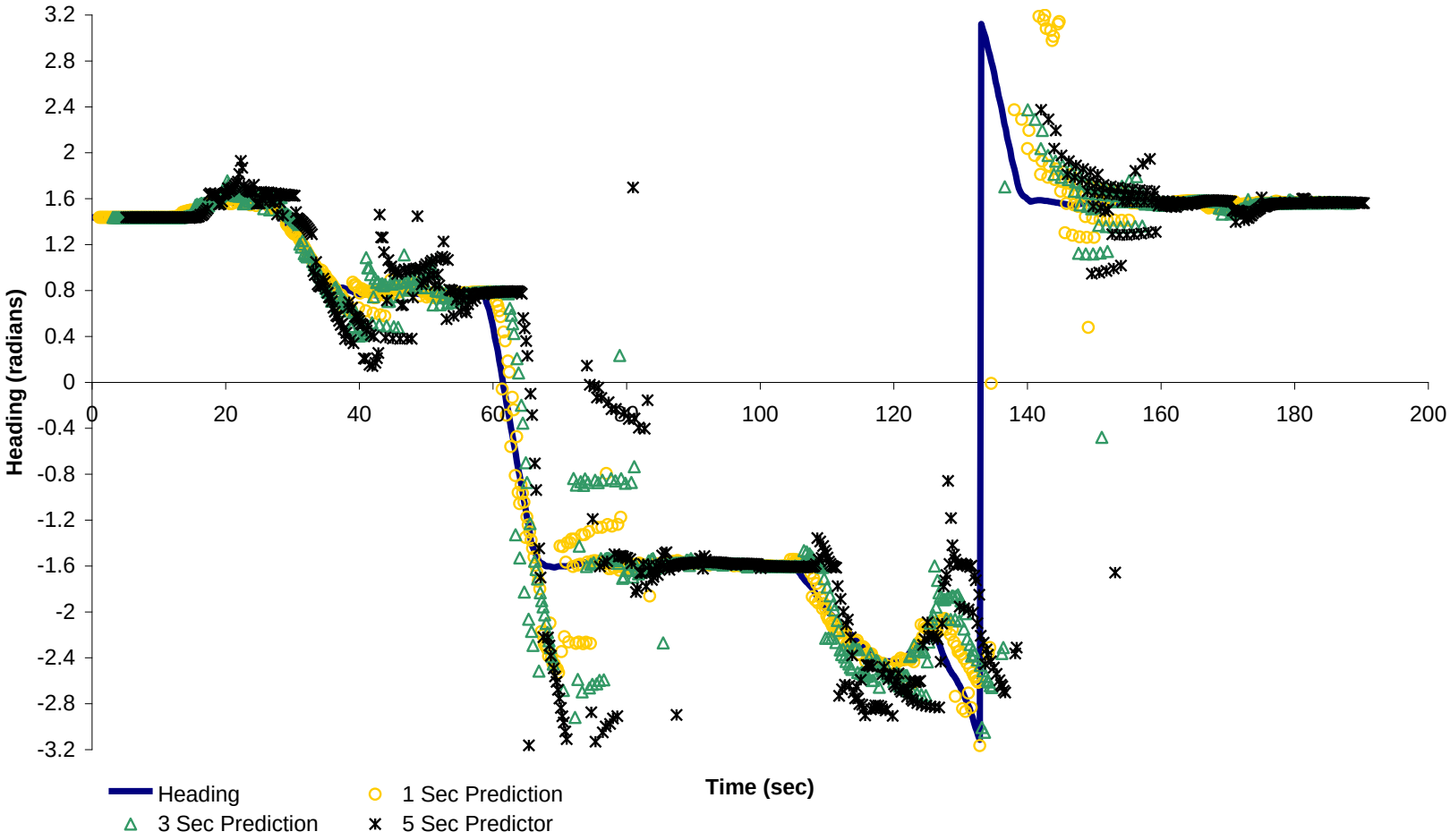

Figure 5-35 Heading vs. Time plot with prediction values for test scenario 5

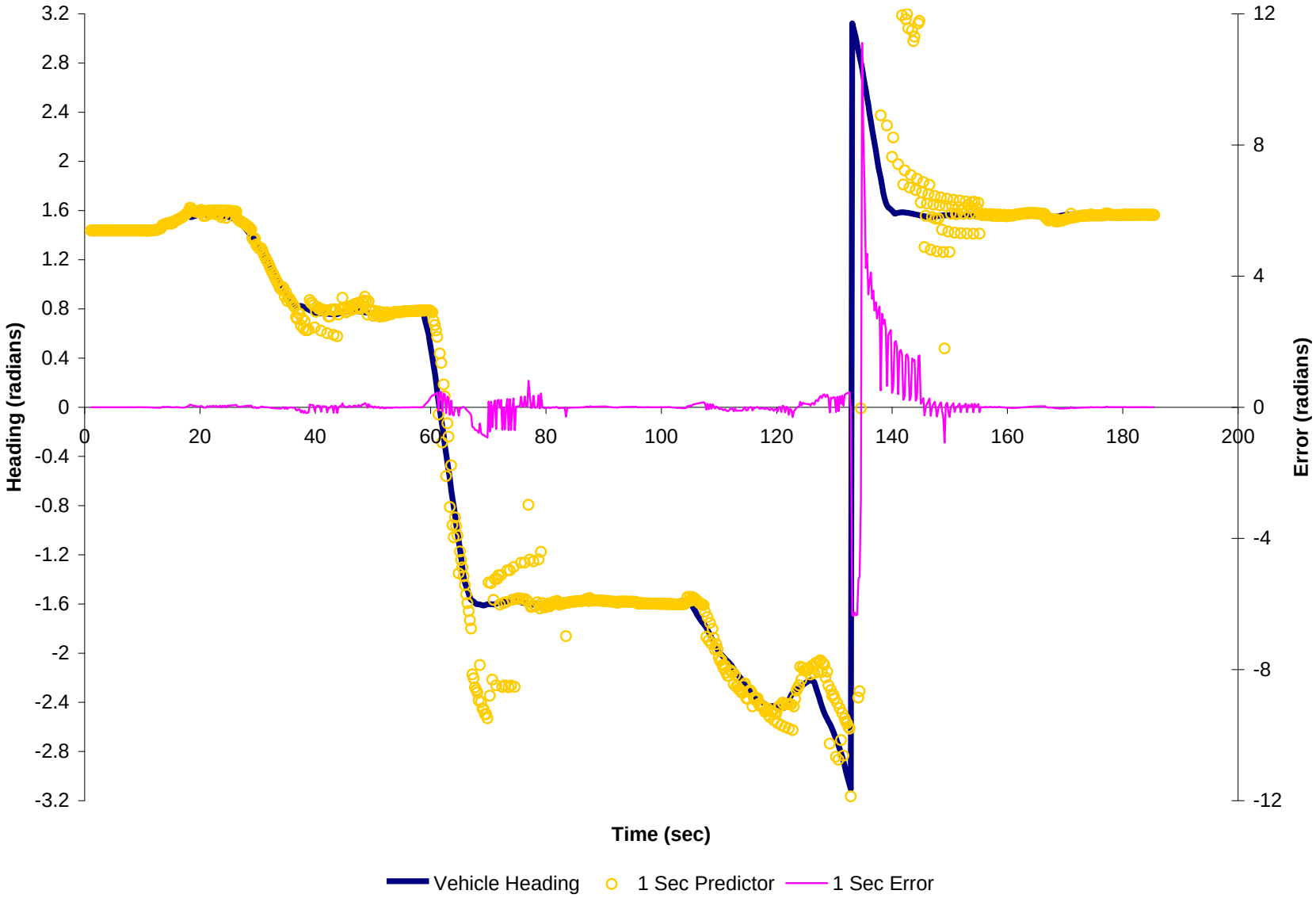

Figure 5-36 Error vs. Time plot for test scenario 5.

CHAPTER 6
FUTURE WORK AND CONCLUSIONS

This dissertation has outlined the author's work to create a centralized knowledge store capable of providing advanced capabilities in coping with the storage and querying of dynamic information. Previous chapters have dealt with the specifics of the knowledge store, its implementation and some preliminary results. In this chapter, the author discusses opportunities for future work in the area and a summary of the dissertation and its contributions to the robotics community as a whole.

Future Work

The author's work has focused on the problem of storing and querying dynamic information in a centralized knowledge store. However, there are a variety of other interesting challenges in the realm of world modeling which are still to be addressed. A significant problem is that of a common taxonomy by which features and objects in the knowledge store can be understood. A main goal of the JAUS architecture is to promote interoperability amongst robotic platforms and between vendors. By adopting a common language and set of features, the information stored in one robot may be more readily understood and used by another.

This alludes to another significant challenge. That is of knowledge store synchronization. In a future world populated by hundreds if not thousands of autonomous and semi-autonomous robotic platforms, sharing information between those platforms will be necessary to success. World Modeling and geospatial information in general have significant challenges when trying to combine information from multiple sources. Questions especially arise when two vehicles have visited the same location but have differing interpretations. Or when visiting a place previously visited by another platform and having a system's sensors interpreting the

environment differently. Who to trust and why become significant questions which must be answered.

The dynamic world model implemented and tested has shown the basic capabilities to handle a large variety of time-variant data. Most of the author's results are based on the Statistics-Based N^{th} Order Polynomial Predictor which was introduced. However, this estimation technique does not take into account any constraints which may be known about the data set. Other, more complex, estimation techniques can be implemented which may yield better results when applied to a specific data set. One technique which holds some promise is to combine state estimation with other knowledge available in the knowledge store; this is often done in the moving objects database field. For example, a vehicle, in most cases, travels along a specific roadway. If knowledge of a road network is available within the knowledge store, the specific geometry of that road a vehicle is on can be used to constrain the solution from the estimator. Other information that may be useful to state estimation techniques is the location of any objects or obstacles and the location of other known robots in the case of a multi-robot system. Other estimation techniques, such as Kalman filters or neural networks, may benefit from more information about the nature of the dataset. For example, while the polynomial predictor shows good results for most of the object-tracking problem, it does exhibit large errors around changes in the data trends. Objects which move more rapidly or in tighter quarters could demonstrate even more sudden changes. Using a model-based approach might yield better results in these environments through an understanding of the underlying dynamics of the system. This could allow the estimator to anticipate the future state based on the current state and the realm of possible or probable inputs to the system.

The polynomial predictor implemented by the author has areas for future improvement as well. One of the greatest strengths of the author's approach is the predictor's ability to be applied to a large variety of time-variant data. Due to a lack of assumptions about the data or its behavior, the polynomial predictor works simply by observing the trend of a data set and extrapolating an assumption that a given trend will continue for some future time period. However, several key areas in the identification of the appropriate future trend could be improved upon. First, it is hypothesized that the use of the configuration parameters window size and window count could be done away with. Rather than iterate on the data set looking for an appropriate history size, some method could be devised to identify the exact spot in the history to which a polynomial should be fit. The author attempted to solve this same problem by examining the time derivative of the data looking for local minimum and maximum points to indicate significant change in the data. This approach did not yield favorable results for the author, but may still be valid. Other approaches investigated but not test by the author were statistics based to find some point at which the data shows a significant change or variation in trend.

Another key part of the polynomial predictor algorithm is the evaluation of different solutions. With the current approach of analyzing a number of different "windows," the one with the lowest order is selected as discussed previously. The reason the lowest order polynomial is selected is because higher-order polynomials tend to exhibit much larger errors when used for extrapolation. However, it is possible that these higher order polynomials might provide better estimation around trend changes because they address the nonlinearity of the data at those points. Therefore it is hypothesized that some other metric for the evaluation of the appropriate solution could be used and yield better results than the current method.

The current implementation of the dynamic world model does not provide any reasoning or analysis of the estimator solutions. Rather, the solution, as found by a prediction algorithm, is reported as-is. Many times this is the proper behavior. However, some form of oversight or regulation functionality could provide added value to the system. For example, while an object may more hundreds of meters through the course of its observed behavior, it is very often not going to do so instantaneously, or near-instantaneously. The capability in the world model to detect situations where a prediction has a high likelihood of being incorrect could either prevent those situations or at least inform a client that such an error may exist. Similar capabilities could (and perhaps should) be implemented in the prediction algorithms themselves, however building this into the primary world model framework would provide basic oversight to all prediction methods deployed.

Similarly, the current implementation requires that the client which generates the data identify the estimation technique to be used with a particular data set. However, no verification of the appropriateness of the selected technique is done. Adding the ability for the knowledge store to analyze or monitor the accuracy of the estimator, in real-time, may allow the knowledge store to alert clients that make use of that data when an inappropriate or significantly bad estimation technique is being used. Another approach may be to put the onus on the world model to select an appropriate estimation technique from a collection automatically. This would still allow for a large variety of techniques to be deployed for use, but exactly which method to use would be selected by the knowledge store using some criterion or other intelligent reasoning algorithm.

Conclusion

These ideas are only the tip of the iceberg for the future of world modeling. As the capabilities and expectations of robotic systems grow, the need for more accurate and powerful

knowledge about the world around the system grows with it. Each new mission holds unique challenges which robotic engineers must solve.

This dissertation has presented the author's work in developing a new and novel approach to storing and querying the state of dynamic information. It began with some background motivation and a simple problem statement. The world is moving, and the robots in it need to be capable of dealing with dynamic constraints. In Chapter 2, background literature is presented which covers a range of topics from motion planning to spatial databases. Chapter 3 outlined the author's novel architecture used to solve various spatiotemporal problems. Also included in Chapter 3 is a technical discussion of a unique new approach to estimating future state of a continuous time-variant function which works in real-time without *a priori* knowledge of the function or any system models. Chapter 4 provides details on the author's implementation of this solution and Chapter 5 discusses some results obtained through 5 test scenarios.

While the author's work has focused on the tracking and prediction of moving objects as observed by some sensing component, the architecture presented should be capable of handling a large variety of objects and attributes, any of which may be dynamic and changing. The work presented herein provides the start to a whole new series of challenges and solutions that may help propel autonomous and semi-autonomous vehicles out of science fiction and into the real world.

APPENDIX A
JAUS WORLD MODEL MESSAGE SET

To support the author's research, the existing JAUS World Model Vector Knowledge Store message set needed to be modified to include a number of additional fields and parameters. Also, a number of new messages needed to be implemented to take full advantage of the new architecture. What follows is a listing of the complete message set, including pre-existing JAUS messages and any messages created or changed by the author.

Local Data Types:

- JAUS Vector Object

Input Messages:

- Code F021h: Set Vector Knowledge Store Feature Class Metadata
- Code F023h: Terminate Vector Knowledge Store Data Transfer
- Code F120h: Request Vector Knowledge Store Object ID Block
- Code F121h: Create Vector Knowledge Store Object(s)
- Code F122h: Modify Vector Knowledge Store Object(s)
- Code F123h: Delete Vector Knowledge Store Object(s)
- Code F130h: Query Vector Knowledge Store Object(s)
- Code F131h: Query Vector Knowledge Store Feature Class Metadata
- Code F132h: Query Vector Knowledge Store Geospatial Bounds
- Code F133h: Query Vector Knowledge Store Temporal Bounds
- Code F134h: Query Vector Knowledge Store Object(s) Future State

Output Messages:

- Code F320h: Assign Vector Knowledge Store Object ID Block
- Code F321h: Report Vector Knowledge Store Object(s) Creation
- Code F322h: Report Vector Knowledge Store Object(s) Modification
- Code F323h: Report Vector Knowledge Store Object(s) Deletion
- Code F330h: Report Vector Knowledge Store Objects
- Code F331h: Report Vector Knowledge Store Feature Class Metadata
- Code F332h: Report Vector Knowledge Store Geospatial Bounds
- Code F333h: Report Vector Knowledge Store Temporal Bounds
- Code F334h: Report Vector Knowledge Store Object(s) Future State
- Code F424h: Report Vector Knowledge Store Data Transfer Termination

JAUS Vector Object

Most of the Vector Knowledge Store messages deal with the transport or modification of geospatial objects and their attributes. To that end, the author has modified the messages to include a new concept in JAUS, a local data type. The local data type, a Vector Object, consists of a series of message fields which are constant anytime the data type is used. Near future work in JAUS is to implement the complete message set in XML and one feature of the proposed XML schema is to support this kind of defined and reusable local data types. Another unique feature of the Vector Object is that is makes use of a local presence vector. While presence vectors have been used extensively in the JAUS message set in the past, never have they been used to control looping chunks of dynamic code. The Vector Object Presence Vector defines which fields are present in an individual Vector Object just as it would for a message. Table A-1 shows the Vector Object fields.

Code F021h: Set Vector Knowledge Store Feature Class Metadata

The Code F021h: Set Vector Knowledge Store Feature Class Metadata message allows the creation, modification, or deletion of feature class metadata. The format of these metadata is not specified. It is left to the system designer to develop a convention for doing this. Initially these data are to be used by the human operators. In the future a convention may be established. Table A-2 shows the fields contained in this message.

Code F023h: Terminate Vector Knowledge Store Data Transfer

The Code F023h: Terminate Vector Knowledge Store Data Transfer message is a command class message that shall cause the vector knowledge store to immediately terminate the transfer of all current and outstanding data destined to the requesting component. Upon termination, the vector knowledge store shall send the requestor the Code F424h: Report Vector Knowledge Store Data Transfer Termination message.

Code F120h: Request Vector Knowledge Store Object ID Block

The Code F120: Request Vector Knowledge Store Object Id Block message is used to request a collection of unique (and continuous) object ids. The knowledge store will attempt to set aside the requested block side object ids for use only by the requesting component. This allows a component to assign ids to objects itself and guarantee there will not be a conflict in the knowledge store. Table A-3 shows the fields contained in this message.

Code F121h: Create Vector Knowledge Store Object(s)

The Code F121h: Create Vector Knowledge Store Objects message is used to add objects to the Vector Knowledge Store. This message allows multiple vector objects to be created using a single message. Field 1 of this message is the creation message properties. If bit zero is set, then the knowledge store shall return the Code F321h: Report Vector Knowledge Store Object(s) Creation message with the local request identifier specified in Field 2. Field 3 indicates the number of vector objects included in the message. Field 4 is the beginning of the definition of a single vector object. Each vector object is defined as outlined above in the Vector Object local data type. Table A-4 shows the fields contained in this message.

Code F122h: Modify Vector Knowledge Store Object(s)

The Code F122h: Modify Vector Knowledge Store Objects message is used to modify existing objects in the Vector Knowledge Store. This message allows multiple vector objects to be modified using a single message. Field 1 of this message is the message properties. If bit zero is set, then the knowledge store shall return the Code F322h: Report Vector Knowledge Store Object(s) Modification message with the local request identifier specified in Field 2. Field 3 indicates the number of vector objects included in the message. Field 4 is the beginning of the definition of a single vector object. Each vector object is defined as outlined above in the Vector

Object local data type. Note that in this message, the Object Id field of each Vector Object is not optional and must be included. Table A-5 shows the fields contained in this message.

Code F123h: Delete Vector Knowledge Store Object(s)

The Code F123h: Delete Vector Knowledge Store Object(s) message is used by a requesting component to delete objects from the Vector Knowledge Store. This message allows for a large number of objects to be removed from the knowledge store. It specifies the objects for removal in one of three ways; specific object id, object attributes or deletion region. Table A-6 shows the fields contained in this message.

Code F131h: Query Vector Knowledge Store Feature Class Metadata

The Code F131h: Query Vector Knowledge Store Feature Class Metadata message shall cause the Vector Knowledge Store to reply to the requestor with the Code F331h: Report Vector Knowledge Store Feature Class Metadata message with the requested data. There is a single field associated with this message. This field specifies the feature class metadata to return in the reply. There is also an option to return metadata for all feature classes present in the queried vector knowledge store. Table A-7 shows the fields contained in this message.

Code F130h: Query Vector Knowledge Store Object(s)

The Code F130h: Query Vector Knowledge Store Objects message allows the access to objects within the vector knowledge store. Table A-8 shows the fields contained in this message.

Code F132h: Query Vector Knowledge Store Geospatial Bounds

The Code 2A22h: Query Vector Knowledge Store Geospatial Bounds message is used to request the spatial extents of a single feature class or of all feature classes within a vector knowledge store. The knowledge store shall respond with the Code 4A23h: Report Vector Knowledge Store Geospatial Bounds message. The boundary is represented by two points which

represent the rectangular region that just covers all of the data within the feature class layer or layers. Table A-9 shows the fields contained in this message.

Code F133h: Query Vector Knowledge Store Temporal Bounds

The Code F133h: Query Vector Knowledge Store Temporal Bounds message is used to request the temporal extents of a single feature class or of all feature classes within a vector knowledge store. The knowledge store shall respond with the Code F433h: Report Vector Knowledge Store Temporal Bounds message. Table A-10 shows the fields contained in this message.

Code F134h: Query Vector Knowledge Store Object Future State

The Code F134h: Query Vector Knowledge Store Object Future State message shall cause the Vector Knowledge Store to reply to the requestor with the Code F334h: Report Vector Knowledge Store Object Future State message with the requested data. Note: Relative Time is a time difference from the time the Vector Knowledge Store receives the message. Table A-11 shows the fields contained in this message.

Code F320h: Assign Vector Knowledge Store Object ID Block

The Code F320h: Assign Vector Knowledge Store Object Id Block message shall be sent by the knowledge store in response to a Code F120h: Request Vector Knowledge Store Object Id Block message. It contains the Local Request ID of the originating request, along with the upper and lower object ID blocks. This block is considered continuous between the two bounds returned. Values given are inclusive. Table A-12 shows the fields contained in this message.

Code F321h: Report Vector Knowledge Store Object(s) Creation

The Code F321h: Report Vector Knowledge Store Object Creation message is used to confirm creation of objects in the vector knowledge store. This message is sent only when an object creation message is requested by setting bit zero in the Code F121h: Create Vector

Knowledge Store Object message. If this bit is set, this message will be transmitted and the local request identifier (field 1) is set to the value sent with the Code F121h: Create Vector Knowledge Store Object message. Table A-13 shows the fields contained in this message.

Code F322h: Report Vector Knowledge Store Object(s) Modification

The Code F322h: Report Vector Knowledge Store Object(s) Modification message is used to confirm modification of the specified objects in the vector knowledge store. This message is only sent when confirmation is requested in the Code F122h: Modify Vector Knowledge Store Object(s) message. The local request identifier is the one included in the original modification message. Each object included in the modification is reported on with a success or failure of the request. Table A-14 shows the fields contained in this message.

Code F323h: Report Vector Knowledge Store Object(s) Deletion

The Code F323h: Report Vector Knowledge Store Object(s) Deletion message is used to confirm deletion of objects from the vector knowledge store. This message is only sent when confirmation is requested in the Code F123h: Delete Vector Knowledge Store Object(s) message. The local request identifier is the one included in the original message. The object ID of each object removed from the knowledge store is returned. The deletion result field is used to indicate success or failure of the overall deletion request and indicates failure if the original request was improper in any form. Table A-15 shows the fields contained in this message.

Code F330h: Report Vector Knowledge Store Objects

The Code F330h: Report Vector Knowledge Store Objects message is sent in direct response to a Code F130h: Query Vector Knowledge Store Objects message. Table A-16 shows the fields contained in this message.

Code F331h: Report Vector Knowledge Store Feature Class Metadata

The Code F331h: Report Vector Knowledge Store Feature Class Metadata message allows access to feature class metadata stored within the vector knowledge store. It is transferred in response to the Code F131h: Query Vector Knowledge Store Feature Class Metadata message. If the query message requests all feature classes, a separate message shall be sent for each feature class. These metadata are entered using the Code F021h: Set Vector Knowledge Store Feature Class Metadata message. Table A-17 shows the fields contained in this message.

Code F332h: Report Vector Knowledge Store Geospatial Bounds

The Code F332h: Report Vector Knowledge Store Geospatial Bounds message format is shown below. This message reports the bounds as a response to the Query Vector Knowledge Store Bounds message. In this message, the knowledge store returns the two geographic points that represent the extents of the data within a feature class layer or all feature class layers. Table A-18 shows the fields contained in this message.

Code F333h: Report Vector Knowledge Store Temporal Bounds

The Code F333h: Report Vector Knowledge Store Temporal Bounds message format is shown below. This message reports the bounds as a response to the Query Vector Knowledge Store Temporal Bounds message. Table A-19 shows the fields contained in this message.

Code F334h: Report Vector Knowledge Store Object(s) Future State

The Code F334h: Report Vector Knowledge Store Object(s) Future State message is sent in response to a Code F134h: Query Vector Knowledge Store Object Future State message. It includes the objects for which the future state was queried along with the results. Table A-20 shows the fields contained in this message.

Code F424h: Report Vector Knowledge Store Data Transfer Termination

The Code F424h: Report Vector Knowledge Store Data Transfer Termination message notifies other JAUS components that data that were being transferred or were going to be transferred to them has been stopped. This message is sent in response to the Code F023h: Terminate Vector Knowledge Store Data Transfer message. It is also sent whenever data transfer is interrupted due to a change in the component state.

Table A-1 JAUS Vector Object Mapping

Field #	Name	Type	Units	Interpretation	PV Bit #
	Object Presence Vector	Short	See Mapping	Bit Field (See column for Object PV Mapping)	
	Object ID	Unsigned Integer	N/A	0x00000000 – Reserved	0
	Object Type	Byte	N/A	Enumeration 0: Point 1: Line 2: Polygon 3 – 255: Reserved	
	Object Buffer	Float	Meters		1
	Object Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: Reserved	2
	Object Feature Class Count (*m*)	Unsigned Short Integer	N/A	Note: If this field is Presence Vectored out, it will be assumed to be equal to 1	3
	Object Feature Class *1* Id	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – Reserved	
	Object Feature Class *1* Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: Reserved	4
	Object Feature Class *1* Attribute Data Type	Byte	N/A	Enumeration 0: Byte 1: Short Integer 2: Integer 3: Long Integer 4: Unsigned Short Integer 5: Unsigned Integer 6: Unsigned Long Integer 7: Float 8: Long Float 9: RGB (3 Bytes) 10 – 255: Reserved	
	Object Feature Class *1* Attribute Value Count (*n*)	Unsigned Integer	N/A	Note: If this field is Presence Vectored out, it will be assumed to be equal to 1	5
	Object Feature Class 1 Attribute Value 1	Byte	Percentage	0-100% confidence in the attribute value	6

Field #	Name	Type	Units	Interpretation	PV Bit #
	Confidence Value				
	Object Feature Class *1* Attribute Value *1*	Varies (see field 4)	Varies with Feature Class		
	Object Feature Class *1* Attribute Value *1* Lower Bound	Varies (see field 4)	Varies with Feature Class	Used to provide a lower bound to the estimated Attribute Value	7
	Object Feature Class *1* Attribute Value *1* Upper Bound	Varies (see field 4)	Varies with Feature Class	Used to provide an upper bound to the estimated Attribute Value	7
	Object Feature Class *1* Attribute Value *1* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	8
	…	…	…	…	
	Object Feature Class 1 Attribute Value *n* Confidence Value	Byte	Percentage	0-100% confidence in the attribute value	6
	Object Feature Class *1* Attribute Value *n*	Varies (see field 4)	Varies with Feature Class		
	Object Feature Class *1* Attribute Value *n* Lower Bound	Varies (see field 4)	Varies with Feature Class	Used to provide a lower bound to the estimated Attribute Value	7
	Object Feature Class *1* Attribute Value *n* Upper Bound	Varies (see field 4)	Varies with Feature Class	Used to provide an upper bound to the estimated Attribute Value	7
	Object Feature Class *1* Attribute Value *n* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range	8

Field #	Name	Type	Units	Interpretation	PV Bit #
				0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	
	…	…	…	…	
	Object Feature Class *m* Id	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – Reserved	
	Object Feature Class *m* Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: Reserved	4
	Object Feature Class *m* Attribute Data Type	Byte	N/A	Enumeration 0: Byte 1: Short Integer 2: Integer 3: Long Integer 4: Unsigned Short Integer 5: Unsigned Integer 6: Unsigned Long Integer 7: Float 8: Long Float 9: RGB (3 Bytes) 10 – 255: Reserved	
	Object Feature Class *m* Attribute Value Count (*n*)	Unsigned Integer	N/A	Note: If this field is Presence Vectored out, it will be assumed to be equal to 1	5
	Object Feature Class *m* Attribute Value *1* Confidence Value	Byte	Percentage	0-100% confidence in the attribute value	6
	Object Feature Class *m* Attribute Value *1*	Varies (see field 4)	Varies with Feature Class		
	Object Feature Class *m* Attribute Value *1* Lower Bound	Varies (see field 4)	Varies with Feature Class	Used to provide a lower bound to the estimated Attribute Value	7
	Object Feature Class *m* Attribute Value	Varies (see field 4)	Varies with Feature Class	Used to provide an upper bound to the estimated Attribute Value	7

Field #	Name	Type	Units	Interpretation	PV Bit #
	1 Upper Bound				
	Object Feature Class *m* Attribute Value *1* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	8
	…	…	…	…	
	Object Feature Class *m* Attribute Value *n* Confidence Value	Byte	Percentage	0-100% confidence in the attribute value	6
	Object Feature Class *m* Attribute Value *n*	Varies (see field 4)	Varies with Feature Class		
	Object Feature Class *m* Attribute Value *n* Lower Bound	Varies (see field 4)	Varies with Feature Class	Used to provide a lower bound to the estimated Attribute Value	7
	Object Feature Class *m* Attribute Value *n* Upper Bound	Varies (see field 4)	Varies with Feature Class	Used to provide an upper bound to the estimated Attribute Value	7
	Object Feature Class *m* Attribute Value *n* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	8
	Object Position Count (*r*)	Unsigned Integer	N/A	Note: If this field is Presence Vectored out, it will be assumed to be equal to 1	9
	Object Position *1* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999	10

Field #	Name	Type	Units	Interpretation	PV Bit #
				Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	
	Object Position *1* Confidence Value	Byte	N/A	0-100% confidence in the position values	11
	Number of Points for Object Position *1* (*p*)	Unsigned Short Integer	N/A		
	Object Position *1* Point *1* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	
	Object Position *1* Point *1* Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	
	Object Position *1* Point *1* Lower Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *1* Point *1* Lower Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	Object Position *1* Point *1* Upper Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *1* Point *1* Upper Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	…	…	…	…	
	Object Position *1* Point *p* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	
	Object Position *1* Point *p* Longitude	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	

Field #	Name	Type	Units	Interpretation	PV Bit #
	(WGS84)				
	Object Position *1* Point *p* Lower Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *1* Point *p* Lower Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	Object Position *1* Point *p* Upper Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *1* Point *p* Upper Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	…	…	…	…	
	Object Position *r* Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	10
	Object Position *r* Confidence Value	Byte	N/A	0-100% confidence in the position values	11
	Number of Points for Object Position *r* (*p*)	Unsigned Short Integer	N/A		
	Object Position *r* Point *1* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	
	Object Position *r* Point *1* Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	
	Object Position *r* Point *1* Lower Bound Latitude	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12

Field #	Name	Type	Units	Interpretation	PV Bit #
	(WGS84)				
	Object Position *r* Point *1* Lower Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	Object Position *r* Point *1* Upper Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *r* Point *1* Upper Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	…	…	…	…	
	Object Position *r* Point *p* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	
	Object Position *r* Point *p* Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	
	Object Position *r* Point *p* Lower Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *r* Point *p* Lower Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12
	Object Position *r* Point *p* Upper Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	12
	Object Position *r* Point *p* Upper Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	12

Table A-2 Code F021h: Set Vector Knowledge Store Feature Class Metadata

Field #	Name	Type	Units	Interpretation
1	Metadata Options	Byte	N/A	Enumeration 0: Append 1: Prepend 2: Overwrite 3 – 254: Reserved 255: Erase All
2	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – Reserved
3	Number of String Characters	Unsigned Short Integer	N/A	0 … 65,535 This field should be equal to zero only when Field 1 is equal to 255 (Erase All)
4	Feature Class Metadata	String	N/A	Variable length string

Table A-3 Code F023h: Request Vector Knowledge Store Object ID Block

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	The local request id is used to identify the response to this message to the originating component.
2	Requested Block Size	Unsigned Integer	N/A	The number of unique object ids being requested.

Table A-4 Code F120h: Create Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation
1	Message Properties	Byte	N/A	Bit Field 0: Request confirmation of object creation 1 – 7: Reserved
2	Local Request ID	Byte	N/A	Request identifier to be used when returning confirmation to requesting component
3	Number of Objects (*n*)	Unsigned Short Integer		0, reserved 1 … 65,535
4	Vector Object 1	JAUS Vector Object	N/A	This is a series of fields which follow the outline given for a JAUS Vector Object. Again, each object has a unique presence vector.
	…	…	…	…
3 + *n*	Vector Object *n*	JAUS Vector Object	N/A	This is a series of fields which follow the outline given for a JAUS Vector Object.

Field #	Name	Type	Units	Interpretation
				Again, each object has a unique presence vector.

Table A-5 Code F122h: Modify Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation
1	Message Properties	Byte	N/A	Bit Field 0: Request confirmation of object modification 1 – 7: Reserved
2	Local Request ID	Byte	N/A	Request identifier to be used when returning confirmation to requesting component
3	Number of Objects (*n*)	Unsigned Short Integer		0, reserved 1 … 65,535
4	Vector Object 1	JAUS Vector Object	N/A	This is a series of fields which follow the outline given for a JAUS Vector Object. Again, each object has a unique presence vector.
	…	…	…	…
3 + *n*	Vector Object *n*	JAUS Vector Object	N/A	This is a series of fields which follow the outline given for a JAUS Vector Object. Again, each object has a unique presence vector.

Table A-6 Code F123h: Delete Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation	PV Bit #
1	Presence Vector	Byte	N/A	See mapping column.	
2	Local Request ID	Byte	N/A	Request identifier to be used when returning data to requesting component	
3	Number of Object IDs (*p*)	Unsigned Short Integer	N/A		1
4	Object ID *1*	Unsigned Integer	N/A		1
	…	…	…	…	
3+*p*	Object ID *p*	Unsigned Integer	N/A		1
4+*p*	Deletion Region	JAUS Vector Object	N/A	JAUS Vector Object which defines the region in which objects can be deleted. Also can specify the feature classes and attribute values which may be deleted.	2

Table A-7 Code F130h: Query Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation	PV Bit #
1	Presence Vector	Unsigned Short	N/A	See mapping column	
2	Response Presence Vector	Byte	N/A	Query Response Presence Vector. See Code F330h: Report Vector Knowledge Store Objects message for Presence Vector format.	
3	Local Request ID	Byte	N/A	Request identifier to be used when returning data to requesting component	
4	History Size	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	0
5	Number of Object IDs (*p*)	Unsigned Short Integer	N/A		1
6	Object ID *1*	Unsigned Integer	N/A		1
…	…	…	…	…	
5+*p*	Object ID *p*	Unsigned Integer	N/A		1
6+*p*	Object Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: All but Static	2
7+*p*	Region Type	Byte	N/A	Enumeration 0: Point 1: Line 2: Polygon 3 – 255: Reserved	3
8+*p*	Region Buffer	Float	Meters		4
9+*p*	Number of Feature Classes (*n*)	Byte	N/A		5
10+*p*	Feature Class *1*	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – All	6

Field #	Name	Type	Units	Interpretation	PV Bit #
11+*p*	Feature Class *1* Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: All but Static	7
12+*p*	Feature Class *1* Attribute Data Type	Byte	N/A	Enumeration 0: Byte 1: Short Integer 2: Integer 3: Long Integer 4: Unsigned Short Integer 5: Unsigned Integer 6: Unsigned Long Integer 7: Float 8: Long Float 9: RGB (3 Bytes) 10 – 255: Reserved	8
13+*p*	Feature Class Attribute *1*	Varies (see field 4)	Varies with Feature Class		8
	…	…	…	…	
10+*p*+3*n*	Feature Class *n*	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – All	6
11+*p*+3*n*	Feature Class *n* Estimation Type	Byte	N/A	Enumeration 0: Static 1 – 254: Algorithm Id 255: All but Static	7
12+*p*+3*n*	Feature Class *n* Attribute Data Type	Byte	N/A	Enumeration 0: Byte 1: Short Integer 2: Integer 3: Long Integer 4: Unsigned Short Integer 5: Unsigned Integer 6: Unsigned Long Integer 7: Float 8: Long Float 9: RGB (3 Bytes) 10 – 255: Reserved	8
13+*p*+3*n*	Feature Class Attribute *n*	Varies (see previous field)	Varies with Feature Class		8
14+*p*+3*n*	Number of Region	Unsigned Short	N/A	0, reserved 1 … 65,535	9

Field #	Name	Type	Units	Interpretation	PV Bit #
	Points (*m)*	Integer			
15+*p*+3*n*	Query Region Point *1* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	9
16+*p*+3*n*	Query Region Point *1* Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	9
	…	…	…	…	
	Query Region Point *m* Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90	9
	Query Region Point *m* Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180	9

Table A-8 Code F131h: Query Vector Knowledge Store Feature Class Metadata

Field #	Name	Type	Units	Interpretation
1	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – All

Table A-9 Code F132h: Query Vector Knowledge Store Geospatial Bounds

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Request identifier to be used when returning data to requesting component
2	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – All Feature Classes

Table A-10 Code F133h: Query Vector Knowledge Store Temporal Bounds

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Request identifier to be used when returning data to requesting component
2	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation. 65,535 – All Feature Classes

Table A-11 Code F134h: Query Vector Knowledge Store Object(s) Future State

Field #	Name	Type	Units	Interpretation	PV Bit #
1	Presence Vector	Byte	N/A	See mapping column	
2	Local Request ID	Byte	N/A	Request identifier to be used when returning data to requesting component	
3	Query Type	Byte	N/A	Enumeration 0: Absolute Time 1: Relative Time 2 – 255: Reserved	
4	Query Time	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	
5	Object ID	Unsigned Integer	N/A		
6	Object Confidence Value	Byte	Percent	Interpretation: 95 would be a 95% Confidence Interval (i.e. α=0.05)	1
7	Number of Feature Classes (m)	Unsigned Short Integer	N/A		
8	Feature Class 1	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 – By implementation. 65,535 – All	
9	Feature Class 1 Confidence Interval Percentage	Byte	Percent	Interpretation: 95 would be a 95% Confidence Interval (i.e. α=0.05)	2
	…	…	…		
7 + m	Feature Class m	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 - By implementation.	

Field #	Name	Type	Units	Interpretation	PV Bit #
				65,535 – All	
7 + 2m	Feature Class m Confidence Interval Percentage	Byte	Percent	Interpretation: 95 would be a 95% Confidence Interval (i.e. α=0.05)	

Table A-12 Code F320h: Assign Vector Knowledge Store Object ID Block

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Request identifier sent by requesting component
2	Object ID Block Lower Bound	Unsigned Integer	N/A	Allocated Object ID block, lower bound
3	Object ID Block Upper Bound	Unsigned Integer	N/A	Allocated Object ID block, upper bound

Table A-13 Code F321h: Report Vector Knowledge Store Object(s) Creation

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Local request identifier sent by creating component
2	Number of Object IDs (*p*)	Unsigned Short Integer	N/A	
3	Object ID *1*	Unsigned Integer		0x00000000 Invalid Object ID *This value is used to inform the remote component that, for some reason, the corresponding object could not be created.*
	…	…	…	…
2+*p*	Object ID *p*	Unsigned Integer		0x00000000 Invalid Object ID *This value is used to inform the remote component that, for some reason, the corresponding object could not be created.*

Table A-14 Code F322h: Report Vector Knowledge Store Object(s) Modification

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Local request identifier sent by creating component
2	Number of Object IDs (*p*)	Unsigned Short Integer	N/A	
3	Object ID *1*	Unsigned Integer	N/A	Unique Object ID of the object modified
4	Object 1 Modification Result	Byte	N/A	Enumeration: 0: Failure 1: Success 2-255: Reserved

Field #	Name	Type	Units	Interpretation
…	…	…	…	…
2+*p*	Object ID *p*	Unsigned Integer	N/A	Unique Object ID of the object modified
3+*p*	Object *p* Modification Result		N/A	Enumeration: 0: Failure 1: Success 2-255: Reserved

Table A-15 Code F323h: Report Vector Knowledge Store Object(s) Deletion

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Local request identifier sent by creating component
2	Deletion Result	Byte	N/A	Enumeration: 0: Failure 1: Success 2-255: Reserved
3	Number of Object IDs (*p*)	Unsigned Short Integer	N/A	This being equal to 0 does not necessarily indicate a failure, as a proper deletion request may still yield 0 objects deleted
4	Object ID *1*	Unsigned Integer	N/A	Unique Object ID of the object deleted
…	…	…	…	…
3+*p*	Object ID *p*	Unsigned Integer	N/A	Unique Object ID of the object deleted

Table A-16 Code F330h: Report Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation	PV Bit #
1	Presence Vector	Byte	N/A	Bit Field: Bit 0: Data is included after field 3. This is based on the presence vector received in the Code F130h: Query Vector Knowledge Store Objects Message. If data are present after field 3, this bit should be set. Bits 1-7: Reserved	
2	Local Request ID	Byte	N/A	Request identifier sent in query message	
3	Number of Objects (*p*)	Unsigned Short Integer	N/A	Number of Objects in Response to Query Message	
4	History Size	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59	1

Field #	Name	Type	Units	Interpretation	PV Bit #
				Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 0...31	
5	Object 1	JAUS Vector Object	N/A	JAUS Vector Object that matches the originating query	1
	…	…		…	
4+*p*	Object *p*	JAUS Vector Object	N/A	JAUS Vector Object that matches the originating query	1

Table A-17 Code F331h: Report Vector Knowledge Store Object(s)

Field #	Name	Type	Units	Interpretation
1	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 – By implementation. 65,535 – Reserved
2	Number of String Characters	Unsigned Short Integer	N/A	
3	Feature Class Metadata	String	N/A	Variable length string

Table A-18 Code F332h: Report Vector Knowledge Store Geospatial Bounds

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Request identifier sent in query message
2	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 – By implementation. 65,535 – Reserved
3	Southwest Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90
4	Southwest Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180
5	Northeast Bound Latitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -90 Upper Limit = 90
6	Northeast Bound Longitude (WGS84)	Integer	Degrees	Scaled Integer Lower Limit = -180 Upper Limit = 180

Table A-19 Code F333h: Report Vector Knowledge Store Temporal Bounds

Field #	Name	Type	Units	Interpretation
1	Local Request ID	Byte	N/A	Request identifier sent in query message
2	Feature Class	Unsigned Short Integer	N/A	Enumeration 0 … 65,534 – By implementation. 65,535 – Reserved
3	Time Stamp	Unsigned Integer	N/A	Bits 0-9: milliseconds, range 0...999 Bits 10-15: Seconds, range 0...59 Bits 16 – 21: Minutes, range 0...59 Bits 22-26: Hour (24 hour clock), range 0..23 Bits 27-31: Day, range 1...31
4	Date Stamp	Unsigned Short	N/A	Bits 0-4: Day, range 1...31 Bits 5-8: Month, range 1...12 Bits 9 – 15: Year, range 2000...2127 Where 0 is 2000, 1 is 2001, etc.

Table A-20 Code F334h: Report Vector Knowledge Store Object(s) Future State

Field #	Name	Type	Units	Interpretation	PV Bit #
1	Local Request ID	Byte	N/A	Request identifier sent in query message	
2	Number of Objects (*p*)	Unsigned Short Integer	N/A	Number of Objects in Response to Query Message	
3	Object 1	JAUS Vector Object	N/A	JAUS Vector Object that matches the originating query	1
	…	…		…	
2+*p*	Object *p*	JAUS Vector Object	N/A	JAUS Vector Object that matches the originating query	1

APPENDIX B
SAMPLE CODE EXCERPTS

This appendix contains selections from of the code written by the author to test the reference implementation in the World Model Vector Knowledge Store (WMVKS). Functions and data structures are included based on their overall importance to key features of the implementation.

JAUS Vector Object

The JAUS Vector object is implemented as four distinct objects – JausWorldModelVectorObject, JausWorldModelFeatureClass, JausWorldModelFeatureClassAttribute and JausWorldModelSpatialDataSet. The structures are defined below:

```
// JausWorldModelVectorObject
typedef struct
{
   JausUnsignedInteger id;                    // Unique Object Id
   JausShortPresenceVector presenceVector;    // Unique (per-object)
                                              // presence vector for fields
   JausByte type;           // Enumeration, see above
   JausFloat bufferMeters; // Buffer Size in meters
   JausByte estimator;      // Enumeration, see above
   Vector featureClasses;  // Dynamic Array of FeatureClass data
   Vector geospatialData;  // Dynamic Array of VectorDatasets
                           // (position history)
}JausWorldModelVectorObject;

// JausWorldModelFeatureClass
typedef struct
{
   JausUnsignedShort id;   // Enumeration, defined by system

   // String of Metadata information defined for this Feature Class
   char metaData[JAUS_WM_FC_METADATA_STRING_LENGTH];

   // Enumeration, defined as the type of estimation algorithm to be used
   // (0=Static, 1-254=Dynamic, 255=Reserved)
   JausByte estimator;

   // List of Feature Class Attributes: Data Type and values
   Vector attributeHistory;

   // List of Feature Class Attributes Lower Bound Values
   Vector ciLowerHistory;
```

```
   // List of Feature Class Attributes Upper Bound Values
   Vector ciUpperHistory;

}JausWorldModelFeatureClass;

// JausWorldModelFeatureClassAttribute
 typedef struct
{
   JausByte dataType;          // Enumeration, see above
   JausTime timeStamp;         // OPTIONAL: Valid time for this FC Attribute
   JausByte confidenceValue;   // OPTIONAL: Confidence value or confidence
                               // interval value (0-100%)
   FeatureClassAttributeValue current;    // Attribute Value
   FeatureClassAttributeValue lowerBound; // OPTIONAL: Lower Bound
   FeatureClassAttributeValue upperBound; // OPTIONAL: Upper Bound
}JausWorldModelFeatureClassAttribute;

// JausWorldModelVectorObjectSpatialDataSet
typedef struct
{
   JausTime timeStamp;         // OPTIONAL: Timestamp for this reading
   PointLla centroid;          // Centroid of this dataset
   JausByte confidenceValue;   // OPTIONAL: confidence interval
   Vector dataPoints;          // PointLLAs in this dataset (Lat & Lon)
   Vector ciUpperPoints;       // OPTIONAL: PointLLAs - Confidence
                               // Interval UpperBounds Points
   Vector ciLowerPoints;       // OPTIONAL: PointLLAs - Confidence
                               // Interval LowerBounds Points
}JausWorldModelVectorObjectSpatialDataSet;
```

Generic Predictor

These structures are used throughout the code to store intermediate results of queries and reports as they are marshaled from the database to the JAUS messages and vice-versa. In the text, the Generic Predictor Interface is discussed. Below is the datastructure of this implementation. Also provided is a sample of the generic toBuffer method and the polynomial predictor's toBuffer solution.

Generic Predictor Structure

```
// Generic Predictor
typedef struct GenericPredictorStruct
{
   // Unique Name
   char name[80];

   // Estimator numeric id
   unsigned int id;
```

```
   // some optimization parameters
   unsigned long minimumPointCount;
   unsigned long maximumPointCount;

   // Current data
   DataPoint *dataArray;
   unsigned long dataSize;

   // Generic handles to Predictor functions
   int (* solver)(struct GenericPredictorStruct *);
   double (* estimate)(struct GenericPredictorStruct *, double);
   int (* solutionToBuffer)(struct GenericPredictorStruct *,
      unsigned char *buffer, unsigned int bufferSizeBytes);
   int (* solutionFromBuffer)(struct GenericPredictorStruct *,
      unsigned char *buffer, unsigned int bufferSizeBytes);
   void (* destroy)(struct GenericPredictorStruct *);

   // Placeholder pointer for specific predictor data
   void *solutionStruct;
}GenericPredictorStruct;
```

Generic Predictor ToBuffer Method

```
// Generic Predictor To Buffer
int genericPredictorSolutionToBuffer(GenericPredictor predictor,
   unsigned char *buffer, unsigned int bufferSizeBytes)
{
   unsigned int index = 0;

   // Pack some predictor values

   // The reason we are packing the estimator id is for idetification
   // later. Because, depending on the dataSize, the object->estimator
   // does not necessarily equal estimator id
   memcpy(buffer+index, &predictor->id, sizeof(unsigned int));
   index += sizeof(unsigned int);

   // Pack the solution struct
   index += predictor->solutionToBuffer(predictor, buffer+index,
      bufferSizeBytes-index);
   return index;
}
```

Polynomial Predictor ToBuffer Method

```
// Polynomial Predictor toBuffer
int polyEstimatorSolutionToBuffer(GenericPredictor polyEst, unsigned char
*buffer, unsigned int bufferSizeBytes)
{
   // We are going to store 3 things: power, beta, and startX
   // This allows us to quickly and easily use the estimator
   unsigned int index = 0;
   PolyEstimatorSolution sol = NULL;
   unsigned int sizeBytes = 0;

   sol = (PolyEstimatorSolution) polyEst->solutionStruct;
```

```
    if(!sol)
    {
        // Bad Pointer
        return 0;
    }

    sizeBytes = ((sol->power+1) * sizeof(double)) + sizeof(int) +
        sizeof(double) + sizeof(unsigned int);

    if(bufferSizeBytes < sizeBytes)
    {
        // This is a problem, not enough space in the buffer
        return 0;
    }

    // Power
    memcpy(buffer+index, &sol->power, sizeof(int));
    index += sizeof(int);

    // startX
    memcpy(buffer+index, &sol->startX, sizeof(double));
    index += sizeof(double);

    // Betas
    memcpy(buffer+index, sol->beta, (sol->power+1) * sizeof(double));
    index += (sol->power+1) * sizeof(double);

    return index;
}
```

Process JAUS Messages

Three particular messages define the behavior of the WMVKS. These are the create, modify and query future state messages. The functions that process these JAUS messages are included below.

Create Objects Message

```
// Process Create
JausBoolean processCreateVksObjectsMessage(JausMessage message)
{
    CreateVksObjectsMessage create = NULL;
    ReportVksObjectsCreationMessage response = NULL;
    JausMessage txMessage = NULL;
    JausWorldModelVectorObject object = NULL;

    unsigned int i = 0;
    unsigned int j = 0;

    int queryResult = 0;

    char tempQuery[4096] = {0};
    char *sqlQuery = NULL;
```

```
create = createVksObjectsMessageFromJausMessage(message);
if(!create)
{
   cError("processCreateVksObjectsMessage:%d: Oops.\n", __LINE__);
   return JAUS_FALSE;
}

if(jausBytePresenceVectorIsBitSet(create->messageProperties,
   VKS_CREATE_PROPERTIES_CONFIRMATION_BIT))
{
   // Confirmation requested
   response = reportVksObjectsCreationMessageCreate();
   response->source->id = wmvks->address->id;
   response->destination->id = create->source->id;
   response->requestId = create->requestId;
   response->objectCount = create->vectorObjects->elementCount;
   response->objectIds = (JausUnsignedInteger *)
      malloc(response->objectCount * JAUS_UNSIGNED_INTEGER_SIZE_BYTES);
}

// Insert Objects into the Data Store
for(i = 0; i < create->vectorObjects->elementCount; i++)
{
   object = (JausWorldModelVectorObject)
            create->vectorObjects->elementData[i];

   if(!jausShortPresenceVectorIsBitSet(object->presenceVector,
      JAUS_WM_OBJECT_PV_OBJECT_ID_BIT))
   {
      // if the object ID bit is not set, we need to retrive the
      // next value from the id_seq so that we can set it to that
      sprintf(tempQuery, "SELECT NEXTVAL('%s_objectid_seq');",
         objectTableName);
      postgresDbQuery(tempQuery);
      object->id =
         atoi(postgresDbGetResultValueByColumnName(0, "nextval"));

      jausShortPresenceVectorSetBit(&object->presenceVector,
         JAUS_WM_OBJECT_PV_OBJECT_ID_BIT);
   }

   sqlQuery = createCreateObjectSql(object);
   queryResult = postgresDbQuery(sqlQuery);

   // done with sqlQuery, so free it
   free(sqlQuery);

   for(j = 0; j < object->featureClasses->elementCount; j++)
   {
      sqlQuery = createCreateFeatureClassSql(
         object->featureClasses->elementData[j], object->id);
      queryResult = postgresDbQuery(sqlQuery);

      // done with sqlQuery, so free it
      free(sqlQuery);
   }
```

```
      if(jausBytePresenceVectorIsBitSet(create->messageProperties,
         VKS_CREATE_PROPERTIES_CONFIRMATION_BIT))
      {
         // Capture Object Id
         if(queryResult == 0)
         {
            // Successfully Inserted
            if(jausShortPresenceVectorIsBitSet(object->presenceVector,
               JAUS_WM_OBJECT_PV_OBJECT_ID_BIT))
            {
               response->objectIds[i] = object->id;
            }
            else
            {
               sprintf(tempQuery, "SELECT CURRVAL('%s_objectid_seq');",
                  objectTableName);
               postgresDbQuery(tempQuery);
               response->objectIds[i] =
                  atoi(postgresDbGetResultValueByColumnName(0, "currval"));
            }
            objectCount++;
         }
         else
         {
            // Unsuccessful
            response->objectIds[i] = 0;
         }
      }
   }

   // Respond if requested
   if(jausBytePresenceVectorIsBitSet(create->messageProperties,
      VKS_CREATE_PROPERTIES_CONFIRMATION_BIT))
   {
      txMessage = reportVksObjectsCreationMessageToJausMessage(response);
      nodeManagerSend(wmvksNmi, txMessage);

      jausMessageDestroy(txMessage);
      reportVksObjectsCreationMessageDestroy(response);
   }

   createVksObjectsMessageDestroy(create);
   return JAUS_TRUE;
}
```

Modify Object Message

```
// Process Modify
JausBoolean processModifyVksObjectMessage(JausMessage message)
{
   ModifyVksObjectMessage modifyMsg = NULL;
   JausWorldModelVectorObject dbObject = NULL;
   JausWorldModelVectorObject msgObject = NULL;
   JausWorldModelVectorObjectSpatialDataSet dataSet = NULL;
```

```
char *sqlQuery = NULL;
char *hexString = NULL;
int queryResult = 0;
unsigned int i = 0;
unsigned int j = 0;

char *dataBuffer = NULL;
unsigned int dataSizeBytes;

JausWorldModelFeatureClass msgFeatureClass = NULL;
JausWorldModelFeatureClass dbFeatureClass = NULL;
JausWorldModelFeatureClassAttribute msgFcAttribute = NULL;
JausWorldModelFeatureClassAttribute dbFcAttribute = NULL;

modifyMsg = modifyVksObjectMessageFromJausMessage(message);
if(!modifyMsg)
{
   cError("processModifyVksObjectMessage:%d: Problem unpacking
      modifyMsg\n", __LINE__);
   return JAUS_FALSE;
}

// For convience, let's grab that object pointer from the message
msgObject = modifyMsg->vectorObject;

// Allocate memory for sqlQuery
sqlQuery = (char *) calloc(1, 1024);

// Now, we want to get the object (if it exists) from the dataBase
sprintf(sqlQuery, "SELECT vectorobject FROM %s WHERE objectid=%d;",
         objectTableName, msgObject->id);
queryResult = postgresDbQuery(sqlQuery);
free(sqlQuery);

// Check the results for errors
if(queryResult != 0 || postgresDbGetResultCount() < 1)
{
   cError("processModifyVksObjectMessage:%d: Problem with modify.
      Object (id=%d) not found in dataBase.\n", __LINE__, msgObject->id);
   modifyVksObjectMessageDestroy(modifyMsg);
   return JAUS_FALSE;
}

// Great! Now we take that object and start comparing

// Get the objectArray so we can create a vector Object
hexString = postgresDbGetResultValueByColumnName(0, "vectorObject");
if(!hexString)
{
   cError("processModifyVksObjectMessage:%d: Problem extracting
      hexString.\n", __LINE__);
   modifyVksObjectMessageDestroy(modifyMsg);
   return JAUS_FALSE;
}

// Now convert our hexString to a VectorObject
dbObject = vectorObjectFromHexString(hexString);
```

```
if(!dbObject)
{
   cError("processModifyVksObjectMessage:%d: Problem extracting
               dbObject.\n", __LINE__);
   modifyVksObjectMessageDestroy(modifyMsg);
   return JAUS_FALSE;
}

// Ok, now we can compare the objects and update parts of the database
// as needed. First let's compare the object types, if the Object Type
// has changed, this could be a big problem. For now, if the type has
// changed, we'll just throw an error since interpreting what to do is
// complicated and unnecessary for this implementation
if(dbObject->type != msgObject->type)
{
   cError("processModifyVksObjectMessage:%d: Problem extracting
      dbObject.\n", __LINE__);
   vectorObjectDestroy(dbObject);
   modifyVksObjectMessageDestroy(modifyMsg);
   return JAUS_FALSE;
}

// Now, if this is a "static" object, we need to replace the
// geospatialData
if(msgObject->estimator == JAUS_WM_ESTIMATOR_STATIC)
{
   // Destroy the dbObject data, we're replacing it
   vectorDestroy(dbObject->geospatialData,
      (void *) vectorObjectDataSetDestroy);

   // Clone the dataSet (should only be one if static) from msgObject
   if(msgObject->geospatialData->elementCount > 1)
   {
      cLog("processModifyVksObjectMessage:%d: WARNING... trying
         to modify a static object and providing more than one set of data
         points.\n");
   }

   dbObject->geospatialData = vectorCopy(msgObject->geospatialData,
      (void *) vectorObjectDataSetCopy,
      (void *) vectorObjectDataSetDestroy);
   if(!dbObject->geospatialData)
   {
      cError("processModifyVksObjectMessage:%d: Problem cloning
         msgObject->geospatialData.\n", __LINE__);
      vectorObjectDestroy(dbObject);
      modifyVksObjectMessageDestroy(modifyMsg);
      return JAUS_FALSE;
   }
}
else
{
   // Else.... dynamic object
   // We need to add the data in the msgObject to the dbObject
   if(msgObject->geospatialData->elementCount > 1)
   {
      cLog("processModifyVksObjectMessage:%d: WARNING... trying
```

```
            to modify an object and providing more than one set
            of data points.\n");
      }
      for(i = 0; i < msgObject->geospatialData->elementCount; i++)
      {
         // BIG WARNING HERE... You REALLY should only modify with one
         // dataset. If we put in duplicate data sets, it could cause big
         // problems for some estimator algorithms. (I think)
         dataSet = vectorObjectDataSetCopy(
            (JausWorldModelVectorObjectSpatialDataSet)
            msgObject->geospatialData->elementData[i]);
         if(!dataSet)
         {
            cError("processModifyVksObjectMessage:%d: Problem cloning
               dataset.\n", __LINE__);
            vectorObjectDestroy(dbObject);
            modifyVksObjectMessageDestroy(modifyMsg);
            return JAUS_FALSE;
         }

         vectorAdd(dbObject->geospatialData, dataSet);
      }
   }

   // Now that our geometry is straight, we should update the geometry object
   // in the dataBase. Note: The geometry object in the db is only the last
   // known true geometry of the object
   hexString = vectorObjectToPostgisGeometryHexString(dbObject);
   if(!hexString)
   {
      cError("createCreateObjectSql:%d: Problem with
         vectorObjectToHexBuffer!\n", __LINE__);
      vectorObjectDestroy(dbObject);
      modifyVksObjectMessageDestroy(modifyMsg);
      return JAUS_FALSE;
   }

   // Allocate memory for sqlQuery
   sqlQuery = (char *) calloc(1, 1024);
   sprintf(sqlQuery, "UPDATE %s SET geom = 'srid=-1; %s'::geometry WHERE
      objectid = %d;", objectTableName, hexString, dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);

   // We're done with this, so free it
   free(hexString);
   free(sqlQuery);

   // Test the result of the query
   if(queryResult != 0)
   {
      cLog("processModifyVksObjectMessage:%d: WARNING... Error when trying to
         update geometry for object (ID=%).\n", dbObject->id);
   }

   // Object Type test & update
   if(dbObject->type != msgObject->type)
   {
```

```
   dbObject->type = msgObject->type;

   // Allocate memory for sqlQuery
   sqlQuery = (char *) calloc(1, 1024);
   sprintf(sqlQuery, "UPDATE %s SET object_type = %d WHERE objectid=%d;",
      objectTableName, dbObject->type, dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);
   free(sqlQuery);
}

// Buffer_meters test & update
if(jausShortPresenceVectorIsBitSet(msgObject->presenceVector,
   JAUS_WM_OBJECT_PV_BUFFER_BIT) && dbObject->bufferMeters !=
   msgObject->bufferMeters)
{
   jausShortPresenceVectorSetBit(&dbObject->presenceVector,
      JAUS_WM_OBJECT_PV_BUFFER_BIT);
   dbObject->bufferMeters = msgObject->bufferMeters;

   // Allocate memory for sqlQuery
   sqlQuery = (char *) calloc(1, 1024);
   sprintf(sqlQuery, "UPDATE %s SET buffer_meters = %f WHERE
      objectid=%d;", objectTableName, dbObject->bufferMeters,
      dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);
   free(sqlQuery);
}

// Estimator test & update
if(jausShortPresenceVectorIsBitSet(msgObject->presenceVector,
   JAUS_WM_OBJECT_PV_OBJECT_ESTIMATOR_BIT) && dbObject->estimator !=
   msgObject->estimator)
{
   cLog("processModifyVksObjectMessage:%d: WARNING... Changing the
      estimator might have strange consequences.\n", __LINE__);
   jausShortPresenceVectorSetBit(&dbObject->presenceVector,
      JAUS_WM_OBJECT_PV_OBJECT_ESTIMATOR_BIT);

   dbObject->estimator = msgObject->estimator;

   // Allocate memory for sqlQuery
   sqlQuery = (char *) calloc(1, 1024);
   sprintf(sqlQuery, "UPDATE %s SET estimator = %d WHERE objectid=%d;",
      objectTableName, dbObject->estimator, dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);
   free(sqlQuery);
}

// Object TimeStamp
// We're going to ASSUME the object timestamp has changed
dataSet = (JausWorldModelVectorObjectSpatialDataSet)
   dbObject->geospatialData->elementData
   [dbObject->geospatialData->elementCount-1];

// Allocate memory for sqlQuery
sqlQuery = (char *) calloc(1, 1024);
sprintf(sqlQuery, "UPDATE %s SET object_timestamp = '%s' WHERE
```

```
   objectid=%d;", objectTableName,
   jausTimeToSqlString(dataSet->timeStamp), dbObject->id);
queryResult = postgresDbQuery(sqlQuery);
free(sqlQuery);

// OK... now we need to update featureClasses
// A couple notes here... first, if a feature class does not match one
// that is already attached to the object, we just add it. Second, if it
// does match, we update the information with the latest attribute value.
// Remember to test if the FC Attribute is static or not
for(i = 0; i < msgObject->featureClasses->elementCount; i++)
{
   msgFeatureClass = (JausWorldModelFeatureClass)
      msgObject->featureClasses->elementData[i];

   for(j = 0; j < dbObject->featureClasses->elementCount; j++)
   {
      dbFeatureClass = (JausWorldModelFeatureClass)
         dbObject->featureClasses->elementData[j];

      if(msgFeatureClass->id == dbFeatureClass->id)
      {
         // ok, we found it the match
         break;
      }

      dbFeatureClass = NULL;
   }

   if(dbFeatureClass != NULL)
   {
      // This is an existing featureClass
      // Let's check some things (like dataType)
      msgFcAttribute = (JausWorldModelFeatureClassAttribute)
                       msgFeatureClass->attributeHistory->elementData[0];
      dbFcAttribute = (JausWorldModelFeatureClassAttribute)
                      dbFeatureClass->attributeHistory->elementData[0];

      if(dbFcAttribute->dataType != msgFcAttribute->dataType)
      {
         cError("processModifyVksObjectMessage:%d: No-no! You cannot
            change the dataType of a feature class!\n", __LINE__);
         continue;
      }

      // Test and update the FC Estimator
      if(jausShortPresenceVectorIsBitSet(msgObject->presenceVector,
         JAUS_WM_OBJECT_PV_FC_ESTIMATOR_BIT) &&
         msgFeatureClass->estimator != dbFeatureClass->estimator )
      {
         cLog("processModifyVksObjectMessage:%d: WARNING... Changing the
            estimator might have strange consequences.\n", __LINE__);
         jausShortPresenceVectorSetBit(&dbObject->presenceVector,
            JAUS_WM_OBJECT_PV_FC_ESTIMATOR_BIT);
         dbFeatureClass->estimator = msgFeatureClass->estimator;
      }
```

```
if(dbFeatureClass->estimator == JAUS_WM_ESTIMATOR_STATIC)
{
   // Static featureClass attribute
   // Therefore replace the value in dbFeatureClass with the one
   // from msgFeatureClass. Destroy the dbFeatureClass data, we're
   // replacing it
   vectorDestroy(dbFeatureClass->attributeHistory,
      (void *) featureClassAttributeDestroy);

   // Clone the dataSet (should only be one if static)
   // from msgObject
   if(msgFeatureClass->attributeHistory->elementCount > 1)
   {
      cLog("processModifyVksObjectMessage:%d: WARNING... trying to
         modify a static attribute and providing more than one set
         of attribute values.\n");
   }

   dbFeatureClass->attributeHistory = vectorCopy(
      msgFeatureClass->attributeHistory,
      (void *) featureClassAttributeCopy,
      (void *) featureClassAttributeDestroy);
   if(!dbFeatureClass->attributeHistory)
   {
      cError("processModifyVksObjectMessage:%d: Problem cloning the
         msgFeatureClass->attributeHistory\n", __LINE__);
      continue;
   }

}
else
{
   // Dynamic Feature Class
   // Add this msgAttribute to dbAttributeHistory
   for(j = 0;
      j < msgFeatureClass->attributeHistory->elementCount;
      j++)
   {
      // BIG WARNING HERE... You REALLY should only modify with one
      // dataset. If we put in duplicate data sets, it could cause
      // big problems for some estimator algorithms. (I think)
      dbFcAttribute = featureClassAttributeCopy(
         (JausWorldModelFeatureClassAttribute)
         msgFeatureClass->attributeHistory->elementData[j]);
      if(!dbFcAttribute)
      {
         cError("processModifyVksObjectMessage:%d: Problem cloning
            msgFeatureClassAttribute.\n", __LINE__);
         continue;
      }

      vectorAdd(dbFeatureClass->attributeHistory, dbFcAttribute);
   }
}

// Allocate memory for sqlQuery
sqlQuery = (char *) calloc(1, 1024);
```

```
            sprintf(sqlQuery, "UPDATE %s SET fc_estimator_type=%d,
               fc_data_type=%d, attribute=%lld, attribute_timestamp='%s' WHERE
               fc_id = %d AND objectid=%d;", fcTableName,
               dbFeatureClass->estimator, dbFcAttribute->dataType,
               dbFcAttribute->current.unsignedLongValue,
               jausTimeToSqlString(dbFcAttribute->timeStamp),dbFeatureClass->id,
               dbObject->id);
            queryResult = postgresDbQuery(sqlQuery);
            free(sqlQuery);

            // Now we'll run the FC Predictor and get its hexBuffer
            hexString = featureClassPredictorToHexBuffer(dbFeatureClass);
            if(!hexString)
            {
               cError("processModifyVksObjectMessage:%d: Problem with
                  featureClassPredictorToHexBuffer!\n", __LINE__);
               vectorObjectDestroy(dbObject);
               modifyVksObjectMessageDestroy(modifyMsg);
               return JAUS_FALSE;
            }

            sqlQuery = (char *) calloc(1, strlen(hexString) + 128);
            sprintf(sqlQuery, "UPDATE %s SET fc_estimator_solution = '%s' WHERE
               fc_id=%d AND objectid=%d;", fcTableName, hexString,
               dbFeatureClass->id, dbObject->id);
            queryResult = postgresDbQuery(sqlQuery);
            free(hexString);
            free(sqlQuery);
         }
         else
         {
            // There was no match for this feature class
            // So we just clone it and add it to the dbObject
            dbFeatureClass = featureClassCopy(msgFeatureClass);
            if(!dbFeatureClass)
            {
               cError("processModifyVksObjectMessage:%d: Problem cloning
                  msgFeatureClass.\n", __LINE__);
               continue;
            }
            vectorAdd(dbObject->featureClasses, dbFeatureClass);

            // Since this is a new feature class, we'll add it to the fc table
            sqlQuery = createCreateFeatureClassSql(dbFeatureClass,
               dbObject->id);
            queryResult = postgresDbQuery(sqlQuery);
            free(sqlQuery);
         }
      }

      // Now we'll put the byteArray version of dbObject back into the DB
      dataSizeBytes = vectorObjectSizeBytes(dbObject);
      dataBuffer = (char *) malloc(dataSizeBytes);
      if(!vectorObjectToBuffer(dbObject, dataBuffer, dataSizeBytes))
      {
         cError("processModifyVksObjectMessage:%d: Problem packing vector
            dbObject to dataBuffer\n", __LINE__);
```

```
      vectorObjectDestroy(dbObject);
      modifyVksObjectMessageDestroy(modifyMsg);
      return JAUS_FALSE;
   }
   hexString = bufferToHex(dataBuffer, dataSizeBytes);

   // Now let's update that into the DB
   sqlQuery = (char *) malloc(strlen(hexString)+128);
   sprintf(sqlQuery, "UPDATE %s SET vectorObject = '%s' WHERE objectid=%d;",
      objectTableName, hexString, dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);

   free(dataBuffer);
   free(hexString);
   free(sqlQuery);

   // Now we need to update the object estimator
   hexString = objectPredictorToHexBuffer(dbObject);
   if(!hexString)
   {
      cError("processModifyVksObjectMessage:%d: Problem with
         featureClassPredictorToHexBuffer!\n", __LINE__);
      vectorObjectDestroy(dbObject);
      modifyVksObjectMessageDestroy(modifyMsg);
      return JAUS_FALSE;
   }

   sqlQuery = (char *) malloc( strlen(hexString) + 128 );
   sprintf(sqlQuery, "UPDATE %s SET estimator_solution = '%s' WHERE
      objectid=%d;", objectTableName, hexString, dbObject->id);
   queryResult = postgresDbQuery(sqlQuery);

   // done with these pointers
   free(hexString);
   free(sqlQuery);

   vectorObjectDestroy(dbObject);
   modifyVksObjectMessageDestroy(modifyMsg);

   modifyCount++;

   return JAUS_TRUE;
}
```

Query Future State Message

```
// Process Query Future State
JausBoolean processQueryFutureStateMessage(JausMessage message)
{
   QueryVksObjectFutureStateMessage query;
   ReportVksObjectFutureStateMessage report;
   JausMessage txMessage;

   JausWorldModelVectorObjectSpatialDataSet dataSet;
   JausWorldModelFeatureClass fcClass;
   JausWorldModelFeatureClassAttribute fcAttribute;
```

```
JausTime queryTime = NULL;
JausTime tempTime = NULL;
int characterCount = 0;

int queryResult;
char sqlQuery[1024] = {0};
char *hexString;
unsigned int i = 0;

GenericPredictor xPredictor;
GenericPredictor yPredictor;
GenericPredictor fcPredictor;

PointUtm pointUtm;
PointLla pointLla;

// Unpack Query
query = queryVksObjectFutureStateMessageFromJausMessage(message);
if(!query)
{
   cError("processQueryFutureStateMessage:%d: Problem unpacking query
      message\n", __LINE__);
   return JAUS_FALSE;
}

// First, let's ask the database for the object
sprintf(sqlQuery, "SELECT object_type, estimator_solution,
   utm_init_longitude FROM %s WHERE objectid=%d;", objectTableName,
   query->objectId);
queryResult = postgresDbQuery(sqlQuery);

// Check the results for errors
if(queryResult != 0 || postgresDbGetResultCount() == 0)
{
   cError("processQueryFutureStateMessage:%d: Object (id=%d) not found in
      dataBase.\n", __LINE__, query->objectId);
   queryVksObjectFutureStateMessageDestroy(query);
   return JAUS_FALSE;
}

// Create report, NOTE: This also creates the report->vectorObject
report = reportVksObjectFutureStateMessageCreate();
if(!report)
{
   cError("processQueryFutureStateMessage:%d: Problem creating report
      message\n", __LINE__);
   queryVksObjectFutureStateMessageDestroy(query);
   return JAUS_FALSE;
}
report->requestId = query->requestId;

// We're going to need one of these, might as well make it now
dataSet = vectorObjectDataSetCreate();
if(!dataSet)
{
   cError("processQueryFutureStateMessage:%d: Error creating dataSet.\n",
```

```
            __LINE__);
        queryVksObjectFutureStateMessageDestroy(query);
        reportVksObjectFutureStateMessageDestroy(report);
        return JAUS_FALSE;
    }
    vectorAdd(report->vectorObject->geospatialData, dataSet);

    // Let's setup the basic reportObjectPresenceVector
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_OBJECT_ID_BIT);
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_FC_COUNT_BIT);
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_ATTRIBUTE_COUNT_BIT);
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_ATTRIBUTE_TIMESTAMP_BIT);
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_POSITION_COUNT_BIT);
    jausShortPresenceVectorSetBit(&report->vectorObject->presenceVector,
        JAUS_WM_OBJECT_PV_POSITION_TIMESTAMP_BIT);

    report->vectorObject->type =
        atoi(postgresDbGetResultValueByColumnName(0, "object_type"));
    report->vectorObject->id = query->objectId;

    // Ok, now we get into the future state prediction

    // First, we should setup our query time. This is based on the query type
    if(query->queryType == VKS_FUTURE_STATE_ABSOLUTE_TIME)
    {
        queryTime = jausTimeCopy(query->queryTime);
        if(!queryTime)
        {
            cError("processQueryFutureStateMessage:%d: Error creating
                queryTime\n", __LINE__);
            queryVksObjectFutureStateMessageDestroy(query);
            reportVksObjectFutureStateMessageDestroy(report);
            return JAUS_FALSE;
        }
    }
    else if(query->queryType == VKS_FUTURE_STATE_RELATIVE_TIME)
    {
        // We're going to use this JausTime object, so we'll need to create it
        tempTime = jausTimeCreate();
        if(!tempTime)
        {
            cError("processQueryFutureStateMessage:%d: Error creating
                queryTime\n");
            queryVksObjectFutureStateMessageDestroy(query);
            reportVksObjectFutureStateMessageDestroy(report);
            return JAUS_FALSE;
        }

        jausTimeSetCurrentTime(tempTime);
        queryTime = jausTimeAddTimeStamp(tempTime, query->queryTime);
        jausTimeDestroy(tempTime);
```

```
    if(!queryTime)
    {
        cError("processQueryFutureStateMessage:%d: Error creating
            queryTime\n", __LINE__);
        queryVksObjectFutureStateMessageDestroy(query);
        reportVksObjectFutureStateMessageDestroy(report);

        return JAUS_FALSE;
    }
}
else
{
    cError("processQueryFutureStateMessage:%d: Unknown query type value!
        (type=%d)\n", query->queryType);
    queryVksObjectFutureStateMessageDestroy(query);
    reportVksObjectFutureStateMessageDestroy(report);
    return JAUS_FALSE;
}

// We'll retrieve the BYTEA of the object estimator from the result of the
// previous query
hexString = postgresDbGetResultValueByColumnName(0, "estimator_solution");

// Take this hexString and shove it! No really, let's do something useful
// with it, like make a couple object predictors
xPredictor = predictorFromHexBuffer(hexString + characterCount,
    &characterCount);
yPredictor = predictorFromHexBuffer(hexString + characterCount,
    &characterCount);

// ok... we should test these
if(!xPredictor || !yPredictor)
{
    cError("processQueryFutureStateMessage:%d: Error creating x and y
        predictors.\n", __LINE__);
    jausTimeDestroy(queryTime);
    queryVksObjectFutureStateMessageDestroy(query);
    reportVksObjectFutureStateMessageDestroy(report);
    return JAUS_FALSE;
}

// We'll need a spot to put these values
pointLla = pointLlaCreate();
if(!pointLla)
{
    cError("processQueryFutureStateMessage:%d: Error creating pointLla.\n",
        __LINE__);
    genericPredictorDestroy(xPredictor);
    genericPredictorDestroy(yPredictor);
    jausTimeDestroy(queryTime);
    queryVksObjectFutureStateMessageDestroy(query);
    reportVksObjectFutureStateMessageDestroy(report);
    return JAUS_FALSE;
}

// We'll also need one of these b/c the estimators are done in UTM space
pointUtm = pointUtmCreate();
```

```
if(!pointUtm)
{
   cError("processQueryFutureStateMessage:%d: Error creating pointUtm.\n",
      __LINE__);
   genericPredictorDestroy(xPredictor);
   genericPredictorDestroy(yPredictor);
   jausTimeDestroy(queryTime);
   queryVksObjectFutureStateMessageDestroy(query);
   reportVksObjectFutureStateMessageDestroy(report);
   return JAUS_FALSE;
}

// Ok... now THIS is our future estimator right here
pointUtm->xMeters = genericPredictorEstimate(xPredictor,
   jausTimeToSeconds(queryTime));
pointUtm->yMeters = genericPredictorEstimate(yPredictor,
   jausTimeToSeconds(queryTime));

// We are done with our predictors, and should destroy them
genericPredictorDestroy(xPredictor);
genericPredictorDestroy(yPredictor);

// We need to transform from UTM to LLA to put this in the message
// Since we don't know what UTM zone this object was in, we can ask the
// DB for the utm_init_longitude and initialize utmLib with that
pointLla->longitudeRadians =
   atof(postgresDbGetResultValueByColumnName(0, "utm_init_longitude"));

// Now we'll call utmInit with that point
utmConversionInit(pointLla);

// We're done with this, so we'll get rid of it
pointLlaDestroy(pointLla);

// OK... now we're free (and safe!) to use utmLib to do the reprojection
pointLla = pointUtmToPointLla(pointUtm);
if(!pointLla)
{
   cError("processQueryFutureStateMessage:%d: Error creating pointLla.\n",
      __LINE__);
   jausTimeDestroy(queryTime);
   queryVksObjectFutureStateMessageDestroy(query);
   reportVksObjectFutureStateMessageDestroy(report);
   return JAUS_FALSE;
}

// Lastly, we'll add our estimate to the reportObject
vectorAdd(dataSet->dataPoints, pointLla);

// One last thing to do is go ahead and put the
// objectTime stamp in the message
jausTimeDestroy(dataSet->timeStamp);
dataSet->timeStamp = jausTimeCopy(queryTime);

// done with our UTM point, let's destroy it
pointUtmDestroy(pointUtm);
```

```
// GREAT! Now we have done the object estimate... we have to do the
// feature class estimates. Luckily, they are easier (I think)
for(i = 0; i < query->featureClassCount; i++)
{
   // First, we'll ask the DB if this object & fcId exists
   // We're only going to need the fc_estimator_solution for each result
   // (which should only be 1!)

   // NOTE: We'll use our "snapshot" view here. This allows us easy access
   // to the JOIN of objectTable and fcTable
   sprintf(sqlQuery, "SELECT fc_id, fc_data_type, fc_estimator_solution
      FROM snapshot WHERE objectid=%d AND fc_id=%d;", query->objectId,
      query->featureClassIds[i]);
   queryResult = postgresDbQuery(sqlQuery);

   // Test the results
   // REALLY only should be 1 result
   if(queryResult != 0 || postgresDbGetResultCount() != 1)
   {
      cError("processQueryFutureStateMessage:%d: FeatureClass (id=%d) for
         object (id=%d) not found in dataBase.\n", __LINE__,
         query->featureClassIds[0], query->objectId);
      continue;
   }

   // Great. Now we need to add a featureClass to our object
   fcClass = featureClassCreate();
   if(!fcClass)
   {
      cError("processQueryFutureStateMessage:%d: FeatureClass
         constructor.\n", __LINE__);
      continue;
   }

   fcAttribute = featureClassAttributeCreate();
   if(!fcAttribute)
   {
      cError("processQueryFutureStateMessage:%d: FeatureClass
         constructor.\n", __LINE__);
      featureClassDestroy(fcClass);
      continue;
   }

   // Stick some basic fc info in here
   fcClass->id = atoi(postgresDbGetResultValueByColumnName(0, "fc_id"));
   fcAttribute->dataType =
      atoi(postgresDbGetResultValueByColumnName(0, "fc_data_type"));

   // Ok.... now we have to get our estimator and run it
   // get the fc_estimator_sol from our db results
   hexString =
      postgresDbGetResultValueByColumnName(0, "fc_estimator_solution");
   characterCount = 0;

   // Take this hexString and ... do something useful with it, like make a
   // fc predictor
   fcPredictor =
```

```
            predictorFromHexBuffer(hexString+characterCount, &characterCount);
        if(!fcPredictor)
        {
            cError("processQueryFutureStateMessage:%d: FeatureClass predictor
                failed.\n", __LINE__);
            continue;
        }

        featureClassAttributeFromDouble(fcAttribute,
            genericPredictorEstimate(fcPredictor,
            jausTimeToSeconds(queryTime)));

        jausTimeDestroy(fcAttribute->timeStamp);
        fcAttribute->timeStamp = jausTimeCopy(queryTime);

        // Add this featureClass and attribute to report->vectorObject
        vectorAdd(fcClass->attributeHistory, fcAttribute);
        vectorAdd(report->vectorObject->featureClasses, fcClass);

        // We're done with this predictor, so we'll get rid of it
        genericPredictorDestroy(fcPredictor);
    }

    // Send the report
    txMessage = reportVksObjectFutureStateMessageToJausMessage(report);
    if(txMessage)
    {
        if(query->scFlag == JAUS_SERVICE_CONNECTION_MESSAGE)
        {
            // Service Connection Query
            txMessage->scFlag = JAUS_SERVICE_CONNECTION_MESSAGE;
            txMessage->sequenceNumber = message->sequenceNumber;
        }
        else
        {
            // Non SC Query
            txMessage->scFlag = JAUS_NOT_SERVICE_CONNECTION_MESSAGE;
            txMessage->sequenceNumber = 0;
        }

        txMessage->source->id = wmvks->address->id;
        txMessage->destination->id = query->source->id;

        nodeManagerSend(wmvksNmi, txMessage);
        jausAddressToString(txMessage->destination, sqlQuery);
        jausMessageDestroy(txMessage);
    }

    // Clean-up
    jausTimeDestroy(queryTime);

    reportVksObjectFutureStateMessageDestroy(report);
    queryVksObjectFutureStateMessageDestroy(query);

    return JAUS_TRUE;
}
```

Prediction Algorithm Solver Functions

The following is the solver functions for the polynomial and linear predictors. These are included to show the details of the SNOPP implementation and how GSL is used.

Linear Predictor

```
// Linear Predictor Solver Function
int linearEstimatorSolver(GenericPredictor linearEst)
{
   // Solution in the form Y = beta0 + beta1 * X

   unsigned int i = 0;
   unsigned int startIndex = 0;

   double *xArray = NULL;
   double *yArray = NULL;

   LinearEstimatorSolution sol = NULL;

   // Solve the problem using the LinearEstimator algorithm
   if(!linearEst)
   {
      cError("linearEstimator:%d: Pointer is NULL.\n", __LINE__);
      return CIMAR_FALSE;
   }

   if(!linearEst->dataArray)
   {
      cError("linearEstimator:%d: Cannot solve because linearEst->dataArray
         is NULL.\n", __LINE__);
      return CIMAR_FALSE;
   }

   sol = (LinearEstimatorSolution) linearEst->solutionStruct;

   if(linearEst->dataSize < sol->confMinimumPointCount)
   {
      cError("linearEstimator:%d: Cannot solve because linearEst->dataSize <=
            sol->confMinimumPointCount (%d < %d).\n", __LINE__,
            linearEst->dataSize, sol->confMinimumPointCount);
      return CIMAR_FALSE;
   }

   if(linearEst->dataSize > sol->confMaximumPointCount)
   {
      startIndex = linearEst->dataSize - sol->confMaximumPointCount;
      linearEst->dataSize = sol->confMaximumPointCount;
   }

   // Allocate memory for the x array
   xArray = (double *) calloc(linearEst->dataSize, sizeof(double));
   if(!xArray)
```

```
    {
        cError("linearEstimator:%d: Cannot solve because error allocating
            memory for xArray.\n", __LINE__);
        return CIMAR_FALSE;
    }

    // Allocate memory for the y array
    yArray = (double *) calloc(linearEst->dataSize, sizeof(double));
    if(!yArray)
    {
        cError("linearEstimator:%d: Cannot solve because error allocating
            memory for yArray.\n", __LINE__);
        free(xArray);
        return CIMAR_FALSE;
    }

    // Setup StartX
    sol->startX = linearEst->dataArray[startIndex].x;

    // Setup the xArray & yArray
    for(i = 0; i < linearEst->dataSize; i++)
    {
        xArray[i] = linearEst->dataArray[startIndex + i].x - sol->startX;
        yArray[i] = linearEst->dataArray[startIndex + i].y;
    }

    gsl_fit_linear(xArray, 1, yArray, 1, linearEst->dataSize, &sol->beta0,
        &sol->beta1, &sol->cov00, &sol->cov01, &sol->cov11, &sol->chiSq);

    free(yArray);
    free(xArray);

    return CIMAR_TRUE;
}
```

Polynomial Predictor

```
// Polynomial Predictor Solver Function
int polyEstimatorSolver(GenericPredictor polyEst)
{
    int i = 0, j = 0, k = 0;
    double yMean = 0;

    // Need storage for interim solutions

    // power values from interim solutions of size confWindowCount
    int *solPower = NULL;
    double *solBeta = NULL;

    // Interim Solution Data
    DataPoint *tempData = NULL;    // subset of dataArray for interim solutions
    int tempDataSize = 0;

    // Solution parameters
    double *pValue;          // need one of these for each confMaximumOrder
    double *tempBeta;        // need confMaximumOrder+1
```

```
double *typeOneSSModel; // need confMaximumOrder+1
double ssError;
double msError;
double yEstimate = 0;

PolyEstimatorSolution sol = NULL;

sol = (PolyEstimatorSolution) polyEst->solutionStruct;

// Solve the problem using the PolyEstimator algorithm
if(!polyEst)
{
   cError("polyEstimator:%d: Pointer is NULL.\n", __LINE__);
   return -1;
}

if(!polyEst->dataArray)
{
   cError("polyEstimator:%d: Cannot solve because polyEst->dataArray is
      NULL.\n", __LINE__);
   return -1;
}

if(polyEst->dataSize < sol->confMinimumPointCount)
{
   cError("polyEstimator:%d: Cannot solve because polyEst->dataSize <
      sol->confMinimumPointCount (%ld < %d).\n", __LINE__,
      polyEst->dataSize, sol->confMinimumPointCount);
   return -1;
}

// Allocate memory for solPower
solPower = (int *) calloc(sol->confWindowCount, sizeof(int));
if(!solPower)
{
   cError("polyEstimator:%d: Problem allocating memory for solPower.\n",
      __LINE__);
   return -1;
}

// Allocate memory for solBeta
// This is [confWindowCount][confMaximumOrder+1]
solBeta = (double *) calloc((sol->confWindowCount *
   (sol->confMaximumOrder+1)), sizeof(double));
if(!solBeta)
{
   free(solPower);
   cError("polyEstimator:%d: Problem allocating memory for solBeta.\n",
      __LINE__);
   return -1;
}

pValue = (double *) calloc(sol->confMaximumOrder+1, sizeof(double));
if(!pValue)
{
   free(solBeta);
   free(solPower);
```

```
   cError("polyEstimator:%d: Problem allocating memory for pValue.\n",
      __LINE__);
   return -1;
}

tempBeta = (double *) calloc(sol->confMaximumOrder+1, sizeof(double));
if(!tempBeta)
{
   free(pValue);
   free(solBeta);
   free(solPower);
   cError("polyEstimator:%d: Problem allocating memory for tempBeta.\n",
      __LINE__);
   return -1;
}

typeOneSSModel =
   (double *) calloc(sol->confMaximumOrder+1, sizeof(double));
if(!typeOneSSModel)
{
   free(tempBeta);
   free(pValue);
   free(solBeta);
   free(solPower);
   cError("polyEstimator:%d: Problem allocating memory for
      typeOneSSModel.\n", __LINE__);
   return -1;
}

// Ok... now we can solve this problem
// First, let go through the data and collect some statistics
// Calculate deltaXMean
sol->deltaXMean = 0;
for(i = 1; i < polyEst->dataSize; i++)
{
   sol->deltaXMean += polyEst->dataArray[i].x - polyEst->dataArray[i-1].x;
}
sol->deltaXMean /= polyEst->dataSize;

// We need to setup sol->power for this analysis
sol->power = sol->confMaximumOrder;

// ********************** NOTE ******************************
// We need to put a check in here to ensure that confWindowStepSize >
// confMaximumOrder. If that is NOT true, we can still do the analysis,
// but we have to skip X generations of the window until tempData >=
// confMaximumOrder. Really we should NOT analyze less that
// PE_MINIMUM_DATA_FACTOR * confMaximumOrder data in a generation
i = 0;
tempDataSize = sol->confWindowStepSize;
while(tempDataSize < sol->confMaximumOrder * PE_MINIMUM_DATA_FACTOR)
{
   i++;
   tempDataSize = sol->confWindowStepSize * i;
}

// Analysis of windowCount
```

```
// NOTE: the i=0 is NOT a bug, it is left out because the loop above sets
// up our i value
for(; i < sol->confWindowCount; i++)
{
   // Allocate memory for tempData
   tempDataSize = sol->confWindowStepSize*(i+1);

   // Need to check if tempDataSize > polyEst->dataSize
   // If this is true, we've run out of data!
   // In this case, we'll run the analysis to the end of the data,
   // then exit
   if(tempDataSize > polyEst->dataSize)
   {
      tempDataSize = polyEst->dataSize;
   }

   // Check if previously allocated
   if(tempData != NULL) free(tempData);

   // this is equal to i * confWindowStepSize
   tempData = (DataPoint *) malloc(sizeof(DataPoint) * tempDataSize);

   // Copy subset from dataArray
   memcpy(tempData, &polyEst->dataArray[polyEst->dataSize-tempDataSize],
               tempDataSize*sizeof(DataPoint));

   // WOO-HOO... let's fit a polynomial
   fitPolynomial(tempData, tempDataSize, sol->confMaximumOrder,
      tempData[0].x, &solBeta[i*sol->confMaximumOrder]);

   // Calculate some statistics parameters
   // Note, right now the only ones acutally needed (or used) are ssError
   ssError = 0;
   for(j=0; j < tempDataSize; j++)
   {
      yMean += tempData[j].y;
      yEstimate = estimate(sol->confMaximumOrder,
         tempData[j].x-tempData[0].x, &solBeta[i*sol->confMaximumOrder]);
      ssError += pow(tempData[j].y - yEstimate, 2);
   }
   yMean /= tempDataSize;

   // Calculate the msError for F-Test statistic
   msError = ssError / (tempDataSize - (sol->confMaximumOrder+1));

   // Now its gets complicated
   // we need to fit a polynomial of each order 1, 2, 3...
   // confMaximumOrder
   for(j = 1; j <= sol->confMaximumOrder; j++)
   {
      // fit a polynomial to the dataset with a power of j
      // put the results in tempBeta
      fitPolynomial(tempData, tempDataSize, j, tempData[0].x, tempBeta);

      // take our newly fit lower-order polynomial and collect some data
      typeOneSSModel[j] = 0;
      for(k = 0; k < tempDataSize; k++)
```

```
            {
               yEstimate = estimate(j, tempData[k].x-tempData[0].x, tempBeta);
               typeOneSSModel[j] += pow(yEstimate - yMean, 2);
            }

            // Type I SS
            // typeOneSSModel[j] - typeOneSSModel[j-1]

            // F-Statistic
            // typeOneSumSquares[j] / msError

            // P Critical Value from F-Test(df1=1, df2=n-(k+1))
            pValue[j] = gsl_cdf_fdist_Q((typeOneSSModel[j] -
               typeOneSSModel[j-1])/msError, 1,
               (tempDataSize-(sol->confMaximumOrder+1)));
         }

         // Now we have to analyze these p-values and find the proper order
         // this is the minimum order we are analyzing
         j = sol->confMinimumOrder;
         while(j < sol->confMaximumOrder && pValue[j+1] <
            sol->confPCriticalValue) j++;
         solPower[i] = j;

         // Usually we would draw conclusions once all the solutions are known
         // But we want the lowest order with the most history (in case of a tie
         // for order). Since the history is increasing and order is being
         // minimized, we can actually do this during evaluation

         // The current solution has an order less (or equal),
         // therefore it is better
         if(solPower[i] <= sol->power)
         {
            sol->power = solPower[i];
            sol->historySize = tempDataSize;
            sol->startX = tempData[0].x;

            // We solve the final polynomial for this dataset
            // with the proper power
            fitPolynomial(tempData, tempDataSize, sol->power,
               tempData[0].x, sol->beta);
         }

         // This is a special check for tempDataSize == polyEst->dataSize
         // This was a sign that we were at the end of the data and need to
         // exit, like now
         if(tempDataSize == polyEst->dataSize)
         {
            break;
         }
      }

      // Free Memory
      free(tempData);
      free(typeOneSSModel);
      free(tempBeta);
      free(pValue);
```

```
   free(solBeta);
   free(solPower);
   return 0;
}

// Polynomial Least-Squares Fit
int fitPolynomial(DataPoint *data, int dataSize, int power, double xOffset,
double *coefficients)
{
   int i, j;
   double chisq = 0.0;
   gsl_matrix *xMatrix = NULL;
   gsl_matrix *covarianceMatrix = NULL;

   gsl_vector *yVector = NULL;
   gsl_vector *solutionVector = NULL;

   gsl_multifit_linear_workspace *workspace = NULL;

   // power + 1 because of constant parameter
   power++;

   // Create an xMatrix of size (dataSize x power)
   xMatrix = gsl_matrix_alloc(dataSize, power);

   // Create the yVector of size (dataSize)
   yVector = gsl_vector_alloc(dataSize);

   // Allocate the solution vector
   solutionVector = gsl_vector_alloc(power);

   // covariance matrix allocation
   covarianceMatrix = gsl_matrix_alloc(power, power);

   // Allocate workspace
   workspace  = gsl_multifit_linear_alloc(dataSize, power);

   // Populate the xMatrix and yVector
   for(i = 0; i < dataSize; i++)
   {
      for(j = 0; j < power; j++)
      {
         gsl_matrix_set(xMatrix, i, j, pow(data[i].x-xOffset, j));
      }

      gsl_vector_set(yVector, i, data[i].y);
   }

   // Least-Squares Fit of data
   gsl_multifit_linear(xMatrix, yVector, solutionVector, covarianceMatrix,
      &chisq, workspace);

   for(i = 0; i < power; i++)
   {
      coefficients[i] = gsl_vector_get(solutionVector,i);
   }
```

```
    // Free allocated resources
    gsl_matrix_free(xMatrix);
    gsl_matrix_free(covarianceMatrix);
    gsl_vector_free(yVector);
    gsl_vector_free(solutionVector);
    gsl_multifit_linear_free(workspace);

    return --power;
}
```

APPENDIX C
NUMERIC EXAMPLE

The purpose of this appendix is to present a numeric example of the Statistics-Based Nth Order Polynomial Predictor (SNOPP) which was presented in Chapter 3. Of the purposes of this example, a known 3rd order polynomial will be used to generate a series of data. A random quantity of error is added to these data points to better represent real world data. The resulting dataset is then evaluated and the resulting polynomial is determined using the SNOPP algorithm.

The following 3rd order polynomial was used to generate the sample dataset: $y = 626 + 0.83x + 1.25x^2 + 0.58x^3$. This was used to generate a dataset with 40 values which are shown in Table C-1. The SNOPP algorithm first looks to fit a high-order polynomial to the dataset. For this example a 5th order polynomial was used. The least-squares method was applied as outlined in chapter 3 (Equation 3-7) and the polynomial solution yielded was $y = 626.629072 - 1.722104x + 3.225149x^2 - 0.009266x^3 + 0.075183x^4 - 0.003471x^5$. The SNOPP algorithm now investigates each of the β values to see if they are statistically significant to the dataset.

To do so, a polynomial of each order up to the 5th order must be calculated using the same least-squares method. This yields a collection of polynomials which can be sequentially tested in a Type I Sum of Squares method. The F-Statistic for each β is calculated as shown in Equation 3-8. First the Mean Squared Error (MSE) of the dataset is calculated using Equation 3-9 as follows:

$$MSE = \frac{\sum_{i=1}^{k}\sum_{j=1}^{n}\left(y_{ij} - \bar{y}\right)^2}{n - (k+1)}$$

$$MSE = \frac{\sum_{i=1}^{k}\sum_{j=1}^{n}\left(y_{ij} - 725.269718\right)^2}{40 - (5+1)}$$

$$MSE = \frac{41.122327}{34}$$

$$MSE = 1.209480$$

Next, the SS_{Model} term is calculated for each β using equation 3-10. The solution for $(SS_{Model})_4$ is shown. The others are done similarly and are listed in table C-2. Note that the $\widehat{\boldsymbol{\beta}}_{\boldsymbol{4}}$ term was found previously by fitting a polynomial to the dataset with a maximum order of 4.

$$(SS_{Model})_4 = \sum_{j=0}^{n} \left(\widehat{\boldsymbol{\beta}}_{\boldsymbol{4}} \widehat{\boldsymbol{x}}_j - \bar{y}\right)^2$$

$$(SS_{Model})_4 = \sum_{j=0}^{n} \left(\begin{bmatrix} 626.324537 \\ -0.300902 \\ 1.884956 \\ 0.457250 \\ 0.007496 \end{bmatrix} \widehat{\boldsymbol{x}}_j - 725.269718 \right)^2$$

$$(SS_{Model})_4 = 445588.491338$$

Finally, the F-Value for each order can be calculated using Equation 3-8. The example below is for the 4th order test. Others are similar and results are included in Table C-2.

$$F_4 = \frac{(SS_{Model})_4 - (SS_{Model})_3}{MSE}$$

$$F_4 = \frac{445588.491338 - 445587.652161}{1.209480}$$

$$F_4 = 0.693833$$

Finally, the P-Critical values are analyzed. Here, the 4th order and higher terms are rejected because they have a P-Critical value above the 0.2 threshold which indicates a greater than 20% chance that those terms are not statistically significant. This indicates the true solution is 3rd order. Therefore the 3rd order polynomial fitted previously is said to be the proper solution. Here the solution is $y = 625.986303 + 0.678063x + 1.303377x^2 + 0.574191x^3$. This is

reasonably close to the original polynomial used to generate the data which was $y = 626 + 0.83x + 1.25x^2 + 0.58x^3$.

Table C-1 Dataset used in numeric example

n	X Value	Y Value	Y Value with Error	Error
1	0	626	626.527643	-0.527643
2	0.2	626.22064	626.641172	-0.420532
3	0.4	626.56912	625.787754	0.781366
4	0.6	627.07328	628.071084	-0.997804
5	0.8	627.76096	627.510589	0.250371
6	1	628.66	627.010799	1.649201
7	1.2	629.79824	629.510619	0.287621
8	1.4	631.20352	630.902289	0.301231
9	1.6	632.90368	631.187051	1.716629
10	1.8	634.92656	634.287322	0.639238
11	2	637.3	638.266131	-0.966131
12	2.2	640.05184	641.14496	-1.09312
13	2.4	643.20992	643.671636	-0.461716
14	2.6	646.80208	645.279907	1.522173
15	2.8	650.85616	652.397536	-1.541376
16	3	655.4	655.518128	-0.118128
17	3.2	660.46144	660.469276	-0.007836
18	3.4	666.06832	665.11101	0.95731
19	3.6	672.24848	671.784873	0.463607
20	3.8	679.02976	678.361989	0.667771
21	4	686.44	686.564124	-0.124124
22	4.2	694.50704	696.33044	-1.8234
23	4.4	703.25872	701.383478	1.875242
24	4.6	712.72288	712.838421	-0.115541
25	4.8	722.92736	723.331611	-0.404251
26	5	733.9	734.933894	-1.033894
27	5.2	745.66864	745.143632	0.525008
28	5.4	758.26112	756.637866	1.623254
29	5.6	771.70528	771.254488	0.450792
30	5.8	786.02896	784.966236	1.062724
31	6	801.26	802.414973	-1.154973
32	6.2	817.42624	815.503091	1.923149
33	6.4	834.55552	835.913328	-1.357808
34	6.6	852.67568	851.049287	1.626393
35	6.8	871.81456	872.889215	-1.074655
36	7	892	891.107437	0.892563
37	7.2	913.25984	911.984246	1.275594
38	7.4	935.62192	934.408955	1.212965
39	7.6	959.11408	959.920287	-0.806207
40	7.8	983.76416	982.771937	0.992223

Table C-2 Statistical Values from numeric example

Order	SS_{Model}	F-Value	P-Critical Value
1	385980.150741	319128.952406	< 0.0001
2	444363.750019	48271.645160	< 0.0001
3	445587.652161	1011.924079	< 0.0001
4	445588.491338	0.693833	0.410677
5	445589.206968	0.591684	0.447080

LIST OF REFERENCES

[1] JAUS, "Reference Architecture Specification ver. 3.2," JAUS Working Group, 2004.

[2] N. Nilsson, "Shakey the Robot," in *Technical Note 325* Menlo Park, CA: SRI International, 1984.

[3] J. Leonard and H. Durrant-Whyte, "Simultaneous map building and localization for an autonomous mobile robot," in *IEEE/RSJ International Workshop on Intelligent Robots and Systems*, 1991, pp. 1442-1447.

[4] C. Wang and C. Thorpe, "Simultaneous Localization and Mapping with Detection and Tracking of Moving Objects," in *IEEE International Conference on Robotics & Automation*, 2002.

[5] M. Montemerlo, S. Thrun, and W. Whittaker, "Conditional Particle Filters for Simultaneous Mobile Robot Localization and People-Tracking," in *IEEE International Conference on Robotics & Automation (ICRA)*, 2002.

[6] D. Hahnel, D. Schulz, and W. Burgard, "Map Building with Mobile Robots in Populated Environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[7] D. Hahnel, R. Triebel, W. Burgard, and S. Thrun, "Map Building with Mobile Robots in Dynamic Environments," in *IEEE International Conference on Robotics and Automation*, 2003.

[8] D. Vasquez, F. Large, T. Fraichard, and C. Laugier, "Moving Obstacles' Motion Prediction for Autonomous Navigation," in *8th International Conference on Control, Automation, Robotics and Vision*, Kunming, China, 2004.

[9] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments," in *IEEE International Conference on Robotics and Automation*, 1994, pp. 3310--3317.

[10] P. Fiorini and Z. Shiller, "Motion Planning in Dynamic Environments Using Velocity Obstacles," *International Journal of Robotics Research*, vol. 17, 1998.

[11] F. Large, S. Sekhavat, Z. Shiller, and C. Laugier, "Towards Real-Time Global Motion Planning in a Dynamic Environment Using the NLVO Concept," in *IEEE International Conference on Intelligent Robots and Systems*, 2002.

[12] S. Petti and T. Fraichard, "Safe Motion Planning in Dynamic Environments," in *International Conference on Intelligent Robots and Systems*, 2005.

[13] J. Berg, D. Ferguson, and J. Kuffner, "Anytime Path Planning and Replanning in Dynamic Environments," in *IEEE International Conference on Robotics and Automation*, 2006.

[14] K. Belghith, F. Kabanza, L. Hartman, and R. Nkambou, "Anytime Dynamic Path-Planning with Flexible Probabilistic Roadmaps," in *IEEE International Conference on Robotics and Automation*, 2006.

[15] J. Albus, "4D/RCS: A Reference Model Architecture for Intelligent Unmanned Ground Vehicles," in *SPIE 16th Annual International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, 2002.

[16] C. Schlenoff, R. Madhavan, and T. Barbera, "A Hierarchical, Multi-Resolutional Moving Object Prediction Approach for Autonomous On-Road Driving," in *IEEE International Conference on Robotics and Automation*, 2004.

[17] C. Schlenoff, R. Madhavan, and Z. Kootbally, "PRIDE: A Hierarchical, Integrated Prediction Framework for Autonomous On-Road Driving," in *IEEE International Conference on Robotics and Automation*, 2006.

[18] I. Dagli, M. Brost, and G. Breuel, "Action Recognition And Orediction for Driver Assistance Systems Using Dynamic Belief Networks," in *Conference on Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, 2002, pp. 179-194.

[19] I. Dagli and D. Reichardt, "Motivation-Based Approach To Behavior Prediction," in *IEEE Intelligent Vehicle Symposium*, 2002.

[20] J. Franke, B. Satterfield, M. Czajkowski, and S. Jameson, "Self-Awareness for Vehicle Safety and Mission Success," in *Unmanned Vehicle System Technology*, 2002.

[21] T. Bandyopadhyay, Y. Li, M. Ang., and D. Hsu, "A Greedy Strategy for Tracking a Locally Predictable Target among Obstacles," in *IEEE International Conference on Robotics and Automation*, 2006.

[22] F. Large, D. Vasquez, T. Fkaichard, and C. Laugier, "Avoiding Cars and Pedestrians Using Velocity Obstacles and Motion Prediction," in *IEEE Intelligent Vehicles Symposium*, 2004.

[23] Y. Sheng and Y. Wu, "Motion Prediction in a high-speed, dynamic environment," in *IEEE International Conference on Tools with Artificial Intelligence*, 2005.

[24] C. Wong, B. Lin, and C. Cheng, "Fuzzy Tracking Method with a Switching Grey Prediction for Mobile Robot," in *IEEE International Conference on Fuzzy Systems*, 2001.

[25] A. Elnagar and A. Hussein, "An Adaptive Motion Prediction Model for Trajectory Planner Systems," in *IEEE International Conference on Robotics &Automation*, 2003.

[26] E. Bevilacqua and T. Kimura, "Obstacle Movement Prediction considering Obstacle's Dynamics and a Priori Knowledge of its Goals," in *IEEE International Conference on Industrial Technology*, 2002.

[27] A. Foka and P. Trahanias, "Predictive Autonomous Robot Navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[28] P. Rigaux, M. Scholl, and A. Voisard, *Spatial Databases: With Application to GIS*. San Francisco: Morgan Kaufmann Publishers, 2002.

[29] OpenGIS Consortium, "OpenGIS Simple Features Specification for SQL," in *OpenGIS Project Document 99-049*: OpenGIS Project Document 99-049, 1999.

[30] R. Güting and M. Schneider, *Moving objects databases*. Amsterdam Boston: Morgan Kaufmann, 2005.

[31] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving Objects Databases: Issues and Solutions," in *10th International Conference on Scientific and Statistical Database Management*, 1998.

[32] O. Wolfson, L. Jiang, A. Sistla, S. Chamberlain, N. Rishe, and M. Deng, "Databases for Tracking Mobile Units in Real Time," in *International Conference on Database Theory*, 1999.

[33] O. Wolfson, H. Cao, H. Lin, G. Trajcevski, F. Zhang, and N. Rishe, "Management of Dynamic Location Information in DOMINO," in *8th International Conference on Extending Database Technology: Advances in Database Technology*, 2002.

[34] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Modeling and Querying Moving Objects," in *IEEE Conference on Data Engineering*, 1997.

[35] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Querying the Uncertain Position of Moving Objects," *Lecture Notes in Computer Science*, vol. 1399, pp. 310-317, 1998.

[36] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain, "Managing uncertainty in moving objects databases," *ACM Trans. Database Syst.*, vol. 29, pp. 463-507, 2004.

[37] A. Meystel and J. Albus, *Intelligent Systems: Architecture, Design, and Control*. New York: Wiley, 2002.

[38] JAUS, "World Model Knowledge Store Components ver. 1.5.1," JAUS Working Group, 2005.

[39] Oracle, "Oracle Spatial User's Guide and Reference, 10g Release 2 (10.2)," 2005.

[40] MapInfo Corporation, "SpatialWare for SQL Server v4.8 User Guide," 2004.

[41] ESRI, "ArcSDE: Advanced Spatial Data Server," 2006.

[42] IBM, "IBM Informix Spatial DataBlade Module User's Guide Version 8.20," 2002.

[43] Refractions Research, "PostGIS Manual v. 1.1.2," 2006.

[44] MySQL, "MySQL 5.1 Reference Manual," 2006.

[45] IBM, "IBM DB2 Spatial Extender and Geodetic Extender User's Guide and Reference, Version 8.2," 2004.

[46] Safe Software, "Feature Manipulation Engine (FME) Readers and Writers," 2005.

[47] OpenGIS Consortium, "Registered Products," 2006.

[48] Refractions Research, "Geometry Engine Open Source," 2006.

[49] GNU, "GNU Science Library," 2007.

[50] B. Touchton, T. Galluzzo, D. Kent, and C. Crane, "Perception and Planning Architecture for Autonomous Ground Vehicles," in *Computer*. vol. 39, December 2006, pp. 40-47.

BIOGRAPHICAL SKETCH

Daniel A. Kent was born on September 20, 1979 in Winter Park, Florida. He was raised in Winter Springs, Florida and graduated from Lyman High School in 1998. He received a B.S. in Aerospace Engineering from Embry-Riddle Aeronautical University in Daytona Beach, Florida in 2002. Mr. Kent is currently a Graduate Research Assistant at the Center for Intelligent Machines and Robotics (CIMAR) at the University of Florida. He is an active member of the Joint Architecture for Unmanned Systems (JAUS) Working Group and the Society of Automotive Engineers (SAE) AS-4 committee, both of which are working to establish a standard focused on the interoperability of robotic systems. He was also a member of Team CIMAR, a participant in the 2004 and 2005 DARPA Grand Challenge competitions, in which the NaviGATOR autonomous ground vehicle placed 8th in the 2004 event and 18th in 2005. After graduation, Mr. Kent will continue his career as an engineer working on unmanned robotic vehicles at Defense Technologies Inc. in Gastonia, NC.